DOCUMENT RESUME

ED 459 703 IR 021 062

AUTHOR	Applin, Elizabeth Anne Gates
TITLE	The Application of Language Acquisition Theory to Programming Concept Instruction: Chunks versus Programs from Scratch.
PUB DATE	1999-12-00
NOTE	121p.; Ph.D., University of Southern Mississippi.
PUB TYPE	Dissertations/Theses - Doctoral Dissertations (041)
EDRS PRICE	EDRS Price MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	Academic Achievement; Community Colleges; *Computer Science Education; Critical Thinking; Instructional Design; *Instructional Effectiveness; Intermode Differences; Language Acquisition; *Programming; *Teaching Methods; Two Year Colleges
IDENTIFIERS	Assignment Models

ABSTRACT

This study explored the relationship between assignment-based teaching methods and achievement in an introductory programming course. Subjects in the study were 42 community college students in south Mississippi. All participants completed the Watson-Glaser Critical Thinking Appraisal and the Productive Environment Preferences Survey. These instruments, along with ACT English and math score and high school math grade point average, formed the pretreatment measures. After two identical programming assignments, students in the control group wrote all programming assignments from scratch while students in the experimental group began adding their coded solutions to larger well-documented programs furnished by the researcher/instructor. Qualitative observations of students from the two groups the following semester indicated that the template methodology could have positive long-term effects on many characteristics of student programs. The programming style, documentation practices, parameter passing mechanism use, code reuse, and modularization practice of those students who had been part of the experimental group was superior to that of the control group participants. Appendices include the student consent statement, evaluation of program quality form, student information survey, comprehensive final exam, programming assignment set, and sample program template. (Contains 36 references.) (Author/MES)

Reproductions supplied by EDRS are the best that can be made
from the original document.

ED 459 703

The Applications of Language Acquisition Theory to Programming Concept Instruction: Chunks versus Programs from Scratch

Elizabeth Anne Gates Applin

Dissertation submitted to the Graduate School of the University of Southern Mississippi in Partial Fulfillment of the Requirements of the Degree of Doctor of Philosophy

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
☒ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

PERMISSION TO REPRODUCE AND DISSEMINATE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY
E. Applin
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
2B

BEST COPY AVAILABLE

IR021062

The University of Southern Mississippi

THE APPLICATION OF LANGUAGE ACQUISITION THEORY TO PROGRAMMING CONCEPT INSTRUCTION: CHUNKS VERSUS PROGRAMS FROM SCRATCH

by

Elizabeth Anne Gates Applin

Abstract of a Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ABSTRACT

THE APPLICATION OF LANGUAGE ACQUISITION THEORY
TO PROGRAMMING CONCEPT INSTRUCTION:
CHUNKS VERSUS PROGRAMS FROM SCRATCH

by

Elizabeth Anne Gates Applin

December 1999

This study explored the relationship between assignment based teaching methods and achievement in an introductory programming course. Subjects in the study were 42 community college students in south Mississippi. All participants completed the Watson-Glaser Critical Thinking Appraisal and the Productive Environment Preferences Survey. These instruments along with ACT English score, ACT math score, and high school math grade point average formed the pretreatment measures for the subjects. After two identical programming assignments, students in the control group wrote all programming assignments from scratch while students in the experimental group began adding their coded

solutions to larger well-documented programs furnished by the researcher / instructor.

Various cognitive psychologists tell us that instructional content should be grouped into chunks, delivered in an optimal sequence, and presented in a context at or just beyond the learner's understanding to force deep processing and permanent memory of the content. Advance organizers allow scaffolding of new ideas onto old established concepts and this introduction presented in advance of the actual topic coverage in a classroom setting has proven effective.

Language acquisition studies indicate that new vocabulary must be introduced in context to aid understanding. Second language acquisition theories also suggest that new material should be presented in forms that are just beyond the learner's current level of competence in the language.

The analysis of covariance statistical procedure was used show effect for group. The adjusted means for the control group and the experimental group on the variable final exam grade were 55.8053 and 62.0232 respectively with an observed significance level of .019. The adjusted means for the variable final course average were 69.8892 and 73.4523 respectively with an observed significance level of .043.

Qualitative observations of students from the two groups in CS2 the following semester seem to indicate that the template methodology can have positive long term effects on many characteristics of student programs. The programming style, documentation practices, parameter passing mechanism use, code reuse, and modularization practice of those students who had been part of the experimental group was superior to that of the control group participants.

COPYRIGHT BY

ELIZABETH ANNE GATES APPLIN

1999

The University of Southern Mississippi

THE APPLICATION OF LANGUAGE ACQUISITION THEORY TO PROGRAMMING CONCEPT INSTRUCTION: CHUNKS VERSUS PROGRAMS FROM SCRATCH

by

Elizabeth Anne Gates Applin

A Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Approved:

_____
Director

_____

_____

_____

_____

_____
Dean of the Graduate School

December 1999

DEDICATION

To my father, Richard Farley Gates, M.D. (1906-1983) whose example of academic excellence brought me inevitably to this point.

To Thomas Higginbotham, for convincing me that going back to school was the right thing to do and that computer science was the place to be.

To James and Grace Hu for laying the solid foundation in programming that all other faculty have built upon.

To my sister, Sarah Frances Gates Sully, for threatening to have A. B. D. put on my headstone.

To my husband, Harold Albert Applin, Jr. whose love, encouragement and support made this whole adventure possible.

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my committee, Dr. Hugh Garraway, Dr. J. T. Johnson, Dr. James E. Miller, Dr. Adel Ali, and Dr. Donald Cotten. Their assistance during the development and completion of this study was invaluable. Each member contributed unique suggestions and critiques to the whole work.

Special thanks to Dr. Garraway, who served as my chairperson. The best lesson of all was learning that I am in charge of my own destiny.

Thanks to Dr. Beth Burchell who provided access to the pretest instruments and information on them. Thanks to Linda Mitchell of the Pearl River Community College Admissions office for her assistance in assembling the student information.

Special thanks also to the 1999 ACM Doctoral Consortium coordinators, Dr. Marian Petre and Dr. Vicki L. Almstrum, and the discussion leaders, Dr. Nell Dale, Dr. Sally A. Fincher, Dr. Ann Fleury, and Dr. Fintan Culwin, for their feedback, critiques, and assistance with finding out of print references. Thanks also to the Consortium participants for their suggestions.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF ILLUSTRATIONS vi

LIST OF TABLES vii

CHAPTER

INTRODUCTION 1

- Statement of the Problem
- Hypothesis
- Justification
- Delimitations
- Assumptions
- Definitions
- Operational Definitions
- Overview

REVIEW OF LITERATURE 12

- Introduction to the Theoretical Framework
- Cognitive-interactionist theory
- Research in Programming Instruction
- Summary

METHODS AND PROCEDURES 24

- Introduction
- Subjects and Population
- Instrumentation
- Design
- Procedures
- Treatment
- Analysis Method

RESULTS 36

- Descriptive Data
- Test of Hypotheses
- Qualitative Discussion and Observations

SUMMARY, CONCLUSIONS, DISCUSSION, LIMITATIONS,
AND RECOMMENDATIONS 57

Summary
Summary of Procedures
Summary of Major Findings
Discussion
Recommendations for Further Study

APPROVALS AND PERMISSIONS 68

THE EVALUATION OF PROGRAM QUALITY 71

STUDENT INFORMATION SURVEY 72

CSC 1613 COMPREHENSIVE FINAL EXAM 73

PROGRAMMING ASSIGNMENT SET 85

SAMPLE PROGRAM TEMPLATE 95

REFERENCES 99

LIST OF ILLUSTRATIONS

Figure Page

1. **Figure 1** Nonequivalent Control Group Design 30

LIST OF TABLES

Table — Page

2. Demographics of the Study Sample Population 38
3. PEPS Variables by Group 39
4. Pretreatment Measures by Group 40
5. Post-treatment Measures by Group 41
6. Test of Hypothesis 1: Differences Between Groups on the Variable of Final Exam 43
7. Test of Hypothesis 2: Differences Between Groups on the Variable of Course Average 44
8. Immediate Environmental Variables by Group 48
9. Emotionality Variables by Group 49
10. Sociological Variables by Group 51
11. Physical Needs Variables by Group 52

CHAPTER I

INTRODUCTION

Acquisition of knowledge and skills is a progressive process. Humans build a knowledge base and associated skills which draw upon it beginning at birth and continuing throughout a lifetime. A child's native tongue is developed progressively and naturally, beginning with crib-talk and language play, progressing to generally recognizable sentence structure, and culminating in the ability to verbalize complex thought. In other areas of human knowledge each bit of new information either refines an established concept or adds a new concept to the knowledge base. This is especially true in the areas of mathematics and science. When these areas are taught most of the formal course structure is a hierarchy of topics and concepts arranged in such a way that each successive course builds on and extends the knowledge of the previous course.

Genishi points out that "children learn the rules of their language at an early age through use and over time without any formal instruction" (Genishi, 1988, p. 1). Much of the way that children use language is directly tied to the examples to which they are exposed. The speech patterns

of parents play a large role in the way children construct their language.

When a student learns language through the traditional educational method for foreign language instruction, it is taught in small pieces. By learning several new words a day and finally by listening to and repeating whole sentences the student becomes more comfortable with the new language. As the lessons progress, the student learns the other forms of the components of the language along with the more complex syntax rules and semantics. Achieving fluency in a spoken language through education takes years of study and practice.

Computer programming skills involve an understanding of the semantics and the syntax of a programming language in addition to problem solving and representation skills. In this respect learning a computer programming language is not unlike learning any human language, but programming languages are very small compared to any spoken language. The programming language considered in this study is C++. The entire C++ language has a total vocabulary of thirty-five "reserved" words. These words can be used in a variety of contexts. Students in an introductory programming course are only asked to learn about twenty of these reserved words and their use within several contexts.

During the course of a normal semester, a beginning

programming student may write as many as ten complete programs of various lengths from perhaps as few as twenty lines to as many as one hundred lines. An instructor usually gives several examples of how one of the words of the language is used, but the students rarely see a well developed program of any real length. Since children learn language by mimicking the use patterns that they experience it may be the case that computer programming students will learn better programming skills by adding to or modifying small specific modules in otherwise well-built programs. This modular approach is used by many faculty, but few studies directly comparing the two teaching methods have been found.

Programming was once something done only by a comparably small group of individuals with certain mathematical talents, but is rapidly becoming a necessary skill for students seeking careers in other disciplines. An entry level programming course is routinely included in the curriculum for engineering, physics, and business students. For this study the course in question involves the practice of problem solving using a particular programming language. In essence two things are being taught simultaneously: (a) how to break a large problem into an organized series of small tasks, (b) how to codify instructions for those tasks in a given programming language.

Statement of the Problem

Is there a difference in the programming style, uses of programming constructs, and depth of understanding of those constructs between students who have written computer programs in chunks and those who have written computer programs from scratch as first course assignments?

The attrition rate in first programming courses is quite high and may be attributable to a lack of understanding of how to put all of the programming language constructs together to form a coded problem solution. In addition to helping students finish this first course, many computer science students 'successfully' finish this first programming course with an incomplete understanding of the concepts that have been covered; this hampers further studies in computer science. As the structures and problems become more and more interdependent and/or abstract, students who have somehow missed an important concept or facet of an earlier course topic may be discouraged from continuing in computer science.

The purpose of the study is to compare student understanding of the individual programming constructs between students who are given a series of well-developed programs which they add to or modify and students who are asked to write a series of entire programs.

Hypothesis

H_1: First semester programming students whose programming assignments are concerned with writing specialized modules that are part of well developed pre-written programs will score differently on the final examination from students who write programs from scratch for each assignment during the semester while controlling for learning style, critical thinking skills, ACT mathematics score, and ACT English score.

H_2: The average final semester grade for the two groups will be different while controlling for learning style, critical thinking skills, ACT mathematics score, and ACT English score.

Justification

Computer science, as a discipline, is young compared to other sciences. The first computer science department was established in 1962. It is probably a safe assumption that computer science educators have had a running debate on what the first course (CS1) should include for at least that long. There is guidance on the national level from the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronic Engineers (IEEE) in the form of curriculum recommendations for both high school and for college. These guidelines are very broad giving topics to be

covered throughout a four year program but not really suggesting when, where or how to cover them. While there is not a singular international course description for CS1, it is likely that all introductory computer programming courses are quite similar.

Very few empirical studies have been done in the area of computer science education and very few have studied teaching methodology at the entry level.

Delimitations

1. The subjects for this study consisted of students attending a small community college in Mississippi and were enrolled in sections of Introduction to Computer Programming I (CSC 1613).

2. The study was limited to the information gathered during the fall semester of 1998 and the spring semester of 1999.

3. The students participating in the study did so as a result of section membership and gave written permission to participate in the data collection and analysis thereof.

4. The stated learning objectives of the programming assignments were the same for both treatments.

5. The programming assignments were typical of those assigned in previous semesters.

6. Other than an introductory orientation lab session

and a special session on debugging, all programming assignments were completed by the students either on home computers or outside of class time in the 'open' laboratory.

7. Although the paid laboratory assistants were asked to withhold help from the programming students for anything other than compiler or computer problems, it cannot be excluded that students received programming help from sources other than the instructor.

8. All variables not specified were considered outside of the scope of this study.

Assumptions

1. It was assumed that all of the measurement instruments used in this study were used in an appropriate manner and that they accurately described student programming performance.

2. It was assumed that the programming assignments used in this study were representative of assignments used in similar courses at other community colleges.

3. It was assumed that the students completed their programming assignments independently.

4. It was assumed that those students identified as novice programmers had little or no experience programming in C++ prior to this study.

Definitions

Advance organizer: introductory material used to bridge the gap between existing knowledge and new material to be presented. In the case where no prior knowledge exists, the advance organizer provides ideational anchorage (Ausubel, 1960).

Algorithm: a step-by-step procedure for solving a problem in a finite amount of time (Dale, Weems & Headington, p. 3).

Compiler: a program that translates a high-level language into machine code (Dale, Weems & Headington, p. A15).

Control structure: a statement used to alter the normally sequential flow of control (Dale, Weems & Headington, p. A15).

Data type: a specific set of data values along with a set of operations on those values (Dale, Weems & Headington, p. A16).

Documentation: the written text and comments that make a program easier for others to understand, use and modify (Dale, Weems & Headington, p. A16).

Dummy parameter: (see formal parameter).

Formal parameter: a variable declared in a function heading (Dale, Weems & Headington, p. A16).

Identifier: a name associated with a function or a data object and used to refer to that function or data object (Dale, Weems & Headington, p. A16).

Iteration: an individual pass through, or repetition of, the body of a loop (Dale, Weems & Headington, p. A17).

Local variable: a variable declared within a block and not accessible outside of that block (Dale, Weems & Headington, p. A17).

Postcondition: an assertion that must be true after execution of a module (Dale, Weems & Headington, p. A17).

Precondition: an assertion that must be true before a module begins executing (Dale, Weems & Headington, p. A17).

Reserved word: a word that has special meaning in C++; it cannot be used as a programmer-defined identifier (Dale, Weems & Headington, p. A18).

Source program: (source code) a program written in a high-level programming language (Dale, Weems & Headington, p. A18).

Operational Definitions

For the purpose of testing the outcomes of the study the following terms will be used.

Achievement in Introduction to Programming I was measured by scores on the comprehensive final examination.

Novice programmer was defined as any student enrolled

in the course CSC 1613, Introduction to Programming I, at Pearl River Community College for the fall semester of 1998 or the spring semester of 1999.

Overview

Chapter II contains a detailed review of the related literature from educational psychology, linguistics, and computer science education.

Chapter III outlines the methodology employed to study the treatment. It also includes a description of the subjects for the study along with pertinent descriptive data on the subjects. A detailed description of all procedures and descriptions of the treatment for the study are included. The instruments employed for measurement of the pretreatment condition and the achievement level are described. The analysis techniques used to examine the data are also detailed.

Chapter IV presents the analysis of the data collected during the study period along with a discussion of the research findings.

Chapter V provides a summary of chapters III and IV as well as suggestions for further research in this area.

While the educational process, both teaching and learning, for other fields of study has been thoroughly researched, studies on how to effectively teach the topics

in a first programming course have just begun.

CHAPTER II

REVIEW OF LITERATURE

The purpose of this chapter is to present a review of the literature related to the questions presented in Chapter I. The research and literature to be explored cover educational psychology, concept acquisition, and linguistic theory as they apply to language learning in a general sense and programming language learning in a more focused sense.

Introduction to the Theoretical Framework

There are two dominant theories of learning in practice today: the "stimulus-response conditioning theories of the neobehavioristic family and the interactionist theories of the cognitive family" (Bigge & Shermis, 1991, p. 43). While differences exist between the two theories there are areas where overlap is apparent.

Cognitive-interactionist theory

Cognitive-interactionist theory is solidly grounded in cognitive psychology. Cognitive psychology is primarily concerned with what we know and how we know it. Pioneers in the area of cognitive psychology are well known to most educators. Dewey, Piaget, Bruner, and Vygotsky were all

BEST COPY AVAILABLE

major contributors to cognitive psychology. Many other researchers have conducted studies reflective of cognitive psychology as the theoretical base. These works include several perspectives on learning that fall under the broader category of cognitive psychology, as well as the more modern cognitive-interactionist psychology. Information processing, levels of processing and subsumption theory are three of these research areas.

The Pioneers of Cognitive Psychology

Dewey's work suggests that knowledge is a series of continuous reconstructions. Piaget's studies centered around how children make sense of their environment and the stages of mental maturation from infancy through an age where critical thinking becomes possible. Vygotsky read Piaget's early work but maintains that language and interaction play a larger role in children's understanding. Bruner's studies expanded on Piaget's work by exploring not just how children think, but also how they learn. Bruner's theories were also influenced by Vygotsky(Howe & Jones, 1998).

Information Processing Theory

Information processing theory was proposed by George Miller. Miller's 1956 article compares the experimental results of several other researchers in the study of memory. He describes a measurement of information and the variance between input (information given) and output (information

retrieved by the subject). By using these metrics, he was able to chart the experimental results of various studies involving single stimulus input (words, tones, etc) and multidimensional stimuli (tone and volume combined, for example).

An analysis of the results revealed that for many systems involving humans and some type of single stimulus information, the accuracy of output information was related to the amount of input information. Miller (1956) describes Pollack's experiments involving unidimensional stimuli such as tones and Beebe-Center, Rogers and O'Connell's experiments on taste intensities. Subjects in these experiments were exposed to various quantities of the input stimuli and asked to identify or recall the input in some manner. Miller compared the data from these experiments and notes that the number of items that could be accurately recalled or accurately identified by a subject is limited. He describes this memory limitation as "... the span of immediate memory ..." (Miller, 1956, p. 90) and explains that it is seven items in length. He also describes experiments with multidimensional stimuli, such as Pollack's experiments combining tone and loudness, which Miller says are more natural stimuli for humans since what we see and hear includes multiple dimensions. By adding dimensions to the stimuli, we can expand this memory limit or "... limit

of absolute judgement ..." from seven to as many as 150 items. "Absolute judgment is limited by the amount of information. Immediate memory is limited by the number of items." (Miller, 1956, p. 92) He further differentiates between what he calls bits of information (unidimensional stimuli) and chunks of information (multidimensional stimuli). Miller is quite clear that only seven (plus or minus two) chunks can be processed at a given time. However, "... the span of immediate memory seems to be almost independent of the number of bits per chunk" (Miller, 1956, p. 93).

This idea of grouping material into chunks for instructional purposes is one of the fundamentals of information processing theory. A chunk can refer to any meaningful unit of information. "Chunking has been verified at all levels of cognitive processing" (Kearsy, 1998).

Levels of Processing

The theoretical framework of processing levels suggests that processing information occurs simultaneously on several levels. The deeper the processing, the more permanent the memory. This theory has been similarly applied in language learning (Kearsy, 1998). The key principles of levels of processing theory are (a) the more processing that occurs during learning, the greater the retention; and (b) processing is automatic unless it is focused on a particular

level (Kearsy, 1998).

Perfetti (1979) reports that a sentence can be more easily recalled by subjects when it is in the context of a paragraph than when it is presented in isolation. This implies that the context forces deeper processing, and therefore, greater retention of information.

Studies have been conducted focusing on various ways of teaching reading to young children identified as "at-risk". Some of these have looked at isolated words versus whole sentence teaching and have concluded that there is not much difference in the two methods. Yet "Schreiber (1980) when describing gains in reading fluency, contended that fluency was attained by much more than decoding single words learned alone; instead, fluency was increased when words were grouped and read in meaningful sequences" (Sevier 1995).

Subsumption Theory

Subsumption theory was proposed by David Ausubel. Subsumption is the act of categorizing a concept into a general cognitive framework (Ausubel, 1963, 1968). Ausubel (1963) contends that there are three phases associated with meaningful learning and retention. The first of these is perception. During this phase, "...attitudes, motivations, expectation, and cultural frames of reference influence learning and retention through such mechanisms as selective emphasis, omission, and distortion." (Ausubel, 1963, p. 50)

During the second phase, the new concept is subsumed by a relevant conceptual system. The third phase involves the application of the concept in its original context or in a new context (Ausubel, 1963).

Another part of Ausubel's work is concerned with the use of advance organizers as tools to introduce a concept within an already familiar framework. Research on the use and efficacy of advance organizers has been conducted by Ausubel (1960, 1968) and Mayer (1979). Mayer's conclusions verified Ausubel's contention that advance organizers have a positive effect on learning outcomes.

Ausubel (1968) contends that the length of time for information remaining in memory is a function of the "...stability and clarity of its anchoring ideas" (p.132). If the information is not stable, newly encountered ideas that are relational cannot be firmly anchored and may actually be confused. Advance organizers help the incorporation of new material in three ways. First, the relevant anchoring concept under which the new material is to be subsumed is brought to the learner's mind. This is in a sense repetition of the previously learned idea which is reinforcement. Next, they provide the optimum anchoring concept which "promotes initial learning and later resistance to obliterative subsumption" (p. 137). Finally, they make most rote memorization unnecessary. This use of

relevant introductory materials presented before the new learning material has proven effective.

An introductory course in computer programming involves two skills: critical thinking / problem solving and the syntax and semantics of a programming language. In 1983, Konvalina, Wileman, and Stephens constructed and tested an aptitude predictor and found that a student's mathematics proficiency is tied to success in computer science courses. This explains the college algebra prerequisite for introduction to computer programming courses at many institutions, but still does not guarantee success in the course. Processes of language acquisition and theories of concept acquisition may provide more insight into student success. The programming knowledge that is learned in the first course should be sufficient enough for students to learn new computer languages more easily and at the same time provide a firm foundation for the next level of programming courses.

Second Language Acquisition Theories

Lightbown and Spada (1995) point out that acquisition of a second language is both helped and hindered by the prior knowledge of how languages work. This prior knowledge can lead the learner to make incorrect guesses about how the

second language works (p. 21). While many of the modern learning theories have been empirically studied with respect to second language acquisition, many have not.

Where second languages are concerned, Chomsky's theories fall into a category known as creative construction theory. Stephen Krashen extended this work in 1982. Krashen's input hypothesis asserts that comprehensible input is necessary for language acquisition and further, that "we acquire by understanding language that contains structure a bit beyond our current level of competence" (p. 21). Krashen also differentiates between language acquisition and language learning. Acquisition, he says, is a subconscious act. Learning on the other hand is a deliberate conscious activity. The input hypothesis puts the emphasis on understanding the meaning of the input with the idea that acquisition of structure happens as a result.

Research in Programming Instruction

Many authors (Alstrum, 1991; Denning, 1989; Konvalina, 1983; Campbell, 1984) have researched the connection between previous mathematical experience and success in computer science courses. The process of programming itself can be thought of as derivation of solutions (Denman, 1994; Dyck, 1993). The evidence is compelling enough for many institutions to require either a combination of high school

mathematics courses and eligibility for calculus I or college algebra as a prerequisite or co-requisite for the CS1 course.

In their 1996 article, Brilliant and Wiseman discussed the impact of using different paradigms (procedural versus functional versus object oriented) and programming languages (Pascal, Modula-2/Ada, C/C++, and Scheme) for the first programming class. They found that the choice of first paradigm has little impact on the student's future course work.

Bailie (1991) conducted a study of novice programmers' use of modularization. The single group of students involved in the study were pre-tested and post-tested. The treatment involved having the students build programs from previously existing modules. Bailie found a significant difference in the use of modules on the posttest which he attributed to the treatment.

Wiedenbeck (1986) conducted a study that investigated the existence of beacons (familiar key lines of code) to aid program comprehension. The study involved 12 novice and 12 experienced programmers. The subjects were given a 23 line undocumented Shellsort program written in Pascal. They were asked both to understand it and to memorize it verbatim and were given a three minute study period to accomplish this. The subjects were then given four minutes in which to recall

as much of the code as possible and three additional minutes to write a description of the function of the program. The three lines that are indicative of a swapping of elements were meant to be a clue (beacon) to general functionality of the program. Her results showed "no indication that novices use beacons in program comprehension". While the experts seemed to recall the program in semantic chunks, the novices recalled the program sequentially and remembered the early lines of code the best. Since the swap lines were deeply embedded in the code, recall of what should have been a beacon was not indicated.

Holt (1987) conducted a study centered around the mental model (cognitive representation)that expert and novice programmers build while performing modification tasks to existing programs. The study involved the use of three different programs each of which was presented in three forms; in-line or sequential, functional decomposition, and object-oriented. "Students recalled more segments or chunks from the functional and object-oriented programs than from the in-line code programs" (p.40). The lack of structure found in sequential programs forces programmers to build their own cognitive model of program function which may or may not be faulty.

In a recent study by Frazer (1998), two assignment based teaching methodologies were utilized in a third year

course, Artificial Intelligence. Programs for this course required the use of a functional programming paradigm using the language LISP. Frazer points out that at this level in her university's program the students have already learned at least two programming languages. She divided the students into two equivalent groups based on their grades in the CS2 course. The treatment involved a control group, which wrote the programming assignments from scratch, and an experimental group, which added code to existing programs that she provided for them. Her results show no significant difference in the two groups of students on the final examination grade or in the quality of a judged assignment. Yet, she related that "It is, however, possible that the teaching approaches may impact differently on the programming skills of first-year students who have little experience" (p. 25). Frazer also comments that differences in student learning styles might be a factor in which teaching methodology leads to greater understanding of the material. This study will investigate the impact of these teaching methodologies on novice programmers.

Summary

Ausubel (1968) discusses the mechanism of concept acquisition as building relationships between established ideas and new concepts. This indicates that a foundation

must be in place for new concepts to be learned. "Most researchers agree that prior knowledge has powerful effects on the learning process" (Chinn & Brewer, 1998, p. 97). This further indicates that there should be a logical sequence to the materials learned. "Where there is some relationship between learning outcomes, sequence becomes essential" (Jones & Russell, 1979, p. 491).

Information should be presented in a meaningful sequence (Jones & Russell, 1979), of carefully constructed chunks (Miller, 1956). New information should be presented in context to force deeper processing and be introduced by an advance organizer that allows subsumption with an already understood concept whenever possible.

CHAPTER III

METHODS AND PROCEDURES

Introduction

This chapter describes the statistical methods used to measure the success of the experiment described in chapter I. The choice of experimental design is discussed followed by description of the population, sampling method, and the description of the sample. The procedures used to collect the data for the experiment is then presented along with details of the instrumentation and treatment. The chapter concludes with a brief discussion outlining the methods for analysis of the data.

Subjects and Population

Students planning to transfer to university degree programs in computer science, physics, and mathematics are required to take Introduction to Programming I. For students interested in any of the engineering or engineering technology fields this course is suggested. The findings in this study should be generalizable to students taking introductory programming courses at the community college level. The sample size was 16 students in the control group

and 26 students in the experimental group.

Sample

The sample that was used for this study consisted of all students enrolled in CSC 1613 Introduction to Computer Programming I during the fall semester of 1998 and the spring semester of 1999. Two sections of CSC 1613 were offered in the fall and a single section in the spring. Students enrolled in the section that best fit their scheduling needs. The treatment was randomly assigned to one section in the fall and was used for the single spring section.

Descriptive Data

Descriptive variable data included the subject's gender, programming experience, age, high school mathematics GPA, college mathematics GPA, final grade in college algebra if available, ACT math score, ACT English score and college GPA when available.

Instrumentation

A pretest measurement is necessary in this research design in order to establish a baseline for the subjects and allow for control of abilities across groups. The use of a pretest can threaten the external validity of the study in that there is the possibility that the pretest might prejudice what is learned by the subjects during the course

of the treatment period (Gay, 1996). It has been the researcher's experience that most pretests given in the mathematics courses at our institution are shortened versions of the final examinations for the course in question. As such they directly represent material to be learned during the semester. These pretests are used for placement in the various mathematics courses from developmental levels through college algebra. It was the researcher's goal, therefore to find a test that would yield a useful measurement without actually measuring ability in a programming language. The content of the pretest did not reflect course content, but rather tested critical thinking ability.

The comprehensive examination in Introduction to Programming I: The final examination used for this study was the same examination that had been used by the researcher / instructor the previous fall semester and was constructed using an assortment of questions from the test data bank provided by the author of the text. This choice was made to prevent bias of this test construction on the part of the researcher / instructor.

The Productivity Environmental Preference Survey (PEPS) by Dunn, Dunn, and Price: This learning style inventory was administered as a pretreatment measure and used as a controlling factor in the analysis of student

achievement based on teaching methodology. PEPS allows for analysis of the conditions under which an adult is likely to learn, problem solve, or be productive. The 20 variables included in the survey are: (a) sound, (b) light, (c) warmth, (d) formal design, (e) motivated/unmotivated, (f) persistent, (g) responsible (conforming), (h) structure, (i) learning alone/peer-oriented, (j) authority-oriented learner, (k) several ways, (l) auditory preferences, (m) visual preferences, (n) tactile preferences, (o) kinesthetic preferences, (p) requires intake, (q) evening/morning, (r) late morning, (s) afternoon, and (t)needs mobility. These variables fall into 4 main categories: immediate environment (a-d), emotionality (e-h), sociological needs (i-k), and physical needs(l-t).

Standard scores on each factor range from 20 to 80 with a mean of 50 and a standard deviation of 10. "Individuals having a standard score of 40 or less or 60 or more find that variable important when they study or work" (Price 1986, p.7).

The PEPS instrument was originally developed in 1986 and has undergone three revisions. Variables were initially identified that "describe the way individuals prefer to learn or work" (Price, 1996, p. 14). The survey is in the form of statements with an associated five level Likert scale response ranging from 'strongly agree' to 'strongly

disagree'. This survey was given to a "non-random sample of 589 adults from several states and from various academic and industrial settings" (Price, 1996, p. 14). From analysis of this survey, thirty-one factors were identified as accounting for 65% of the variance. The current version of the PEPS scores the twenty environmental factors and provides a detailed picture of the optimal learning / working environment for the subject. Reliabilities on the 20 factors ranged from .33 to .91 with ninety percent of them greater than or equal to .60(Price 1996, p. 14). Price reported that a study conducted in 1984 by Guinta "evidenced no relationship between matched teacher/student styles and academic achievement" however, a mismatch of the teacher and student styles contributes to teacher stress (p. 24).

The Watson-Glaser Critical Thinking Appraisal (WGCTA): This standardized test was administered along with the PEPS by the college as part of a college research study and was added to this study because of its widespread use and the amount of data available on its use. Standardized tests tend to be non-specific for specialized skills, however, research indicates that critical thinking ability plays a part in the success of programming students.

Originally developed in 1964, the WGTCA was modified in 1980 into the form used for this study. The test is composed of 5 sub-tests that measure; inference ability, recognition

of unstated assumptions, deduction ability, interpretation skills and argument evaluation. Each sub-test is composed of scenarios accompanied by several questions that the respondent is asked to answer. The composite of the sub-tests has been shown to yield a reliable measure of critical thinking ability. For the development sample the internal consistency was measured at .81 and the test-retest reliability for the sample was .81. No significant differences were observed for males and females(Watson & Glaser).

Design

Experimental research in education is conducted on all educational levels and across multiple disciplines. Unlike many kinds of research, we cannot randomly select our subjects and force them into classrooms for our studies. We must take the samples that present themselves in courses or classrooms and then use statistical methods to control for the elements that are outside of our direct control. Nonequivalent control group design Campbell (1963), explains is "One of the most wide-spread experimental designs in educational research ..." (p. 47). The reason for this is that the groups "... do not have pre-experimental sampling equivalence" (p. 47). The sample represents what Campbell calls "... naturally assembled collectives such as

classrooms ..." (p. 47). The treatment must be assigned to one group or the other in some random way that is under the control of the experimenter.

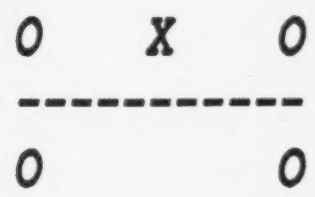


Figure 1 Nonequivalent Control Group Design

Statistical regression is the tendency for the subjects who score at the extremes on the pretest to move closer to the mean on the posttest. If the researcher chooses the lowest pretest scoring subjects for a treatment group, the probability that they will score higher on the posttest is very good regardless of the treatment. While there is a threat to the internal validity of an experiment due to this phenomenon, it has been controlled for in this study due to the nature of the existing groups that are being used.

Selection interaction occurs when the existing groups chosen for a study are very different in characteristics such as an honors course vs a remedial course. This factor has been controlled for in this study, since the two groups represent the same course.

Pretest-treatment interaction occurs when the subjects

are sensitized to the 'important' material during the course of taking the pretest. Posttest scores tend to be higher regardless of the treatment involved. The pretest instruments in this study were specifically chosen because they do not reflect course content thus controlling for this source of invalidity.

Statistical equivalence of the two treatment groups was established through the use of the following controlling variables:

1. Standardized ACT mathematics scores which numerous studies have found to be related to success in computer programming;
2. Standardized English ACT scores which numerous studies have found to be related to success in computer programming;
3. The Productivity Environmental Preference Survey (PEPS), a measure used to determining a teacher's teaching style and a student's learning style;
4. The Watson-Glaser Critical Thinking Appraisal, a standardized test was used as a pretest measure.

This study contained a control group and an experimental group. The experimental group received the independent treatment involving the use of pre-written well-documented programs that were modified or added to by the students as assignments for the course. The control group

was required to write whole programs for their course assignments. The study focused on the effects of using the advance organizer (the pre-written program which contained fully documented elements as yet unfamiliar to the students) as a means of improving the coding style and concept understanding of the students.

The dependent variable, understanding of the course material was measured by the student's score on the final examination which was given to both groups. This examination was evaluated by two expert introductory programming instructors who agreed that the test was an adequate measure of the course content.

Procedures

All students were asked to indicate their agreement to participate in the study by signing a statement. The statement was developed by the researcher in accordance with the guidelines determined by Dean of Academic Affairs at Pearl River Community College. A copy of this statement appears in Appendix A.

Approval for this study was granted by the University of Southern Mississippi through the Human Subjects Protection Review Committee. A copy of this approval appears in Appendix A.

Some descriptive data was obtained from the registrar's

office including gender, age, ACT math score, ACT English score, high school mathematics grade point average, college mathematics grade point average, overall college grade point average and the number of both college level and high school mathematics courses. The high school transcript was used to identify previous programming experience. The final grade for college algebra was used if available. A copy of this form is found in Appendix C.

The treatment period extended over ten weeks of the fall term of 1998 and 10 weeks of the spring term of 1999. One fall term section had scheduled class meetings on Monday, Wednesday, and Friday from 10:00 a.m. to 10:50 a.m. The other fall section and the spring section had scheduled class meetings on Tuesday and Thursday from 8:00 a.m. to 9:15 a.m. Great care was taken to ensure that the weekly instruction was equivalent for both groups. The subjects were all taught by the same instructor. The researcher / instructor has had five years of teaching experience and has taught this particular course each fall. The control group teaching methodology is the method that she has always used. The researcher / instructor is familiar with the experimental treatment and has no personal preference between the two methods beyond that of regarding one as a more traditional method for CS1.

The same textbook *Programming and Problem Solving with C++* by Nell Dale, Chip Weems and Mark Headington was used for all sections. Chapter handouts that contained the text of all of the overhead transparencies used in the lecture were given to all students.

The subjects were given equivalent hands-on instruction in the lab to orient them to the operating system in use, the editor environment and the function of the compiler. A second hands-on session provided the students with instruction on the workings of the debugging environment.

Treatment

The instructional method for both groups was lecture / demonstration done in the classroom except for those lab sessions mentioned above. The first three programming assignments were identical for both groups. From that point forward for the ten-week period, the students in the experimental group were given program templates that they were assigned to either modify or add to. The learning objectives for corresponding assignments were the same for both groups. The seven programming assignment sets appear in Appendix E.

Analysis Method

Data was analyzed using analysis of covariance (ANCOVA) and a significance level of $\alpha = .05$. Analysis of covariance

is used in nonequivalent control group designs to statistically equate two or more groups on some number of variables. The covariate is used to systematically remove "individual differences among subjects from the estimate of experimental error" (Shavelson, 1996, p.504). The only restriction is that the instrument used to measure the covariant must be related to the dependent variable that is being measured (Gay, 1996).

Summary

Since the subjects in this study could not be randomly assigned to the groups, quasi-experimental design was indicated and the non-equivalent control group design was chosen. The independent variable of treatment and control groups was randomly assigned. Instruments for quantifying the dependent variables of final examination grade, and semester grades have been defined. Covariants for the study were identified and descriptives for the study listed.

The treatment consisted of two assignment-based teaching methodologies that have not been tested experimentally at this course level.

In Chapter IV an analysis of the data that was collected is presented along with explanations of all pertinent statistical procedures.

CHAPTER IV

RESULTS

The purpose of this study was to compare student achievement in Introduction to Programming I between students who were given a series of well-developed programs which they added to or modified and students who were asked to write a series of entire programs while controlling for learning style, score on the WGCTA, ACT math, ACT English, and high school mathematics GPA.

The experimental group actually consisted of two separate sections of the course taught in successive semesters. The original experimental group, being an 8:00 a. m. fall semester class, had a very large withdrawal rate (61%) leaving the total number of subjects at nine. Since a careful diary of topics and schedules had been kept by the researcher / instructor, it was possible to recreate the pace of topic coverage and the general atmosphere in the spring semester. There was no statistically significant difference in the two groups on the pre-treatment variables and the decision was made to collapse the groups.

This chapter presents the descriptive data related to the variables of the study and the tests of the hypotheses. A qualitative discussion of observations follows the

quantitative results.

Descriptive Data

The data in Table 1 represent demographic data on the study sample population. A total of 42 subjects was involved in the study. The control group consisted of 16 subjects. The experimental group consisted of 26 subjects. Males made up 68.8% of the control group and 69.2% of the experimental group. Ages in the two groups ranged from 18 to 40 years. All subjects in the control group were white while the experimental group was 88.5% white. With respect to declared major field of study 81.4% of the control group had declared majors in the sciences while only 46.1% of the experimental group had declared majors in the sciences.

Table 1

Demographics of the Study Sample Population

	Control (n=16)		Experimental (n=26)	
	count	%	count	%
Gender				
Male	11	68.8	18	69.2
Female	5	31.3	8	30.8
Race				
White	16	100.0	23	92.9
Black			3	11.5
Major				
Computer Science	5	31.3	6	23.1
Engineering	4	25.0	3	11.5
Engineering Technology	1	6.3	2	7.7
Mathematics	1	6.3	1	3.8
Pre-Med	2	12.5		
Humanities			2	7.7
General Studies	1	6.3	6	23.1
Undeclared	2	12.5	6	23.1
Age				
18 - 19	8	50.1	8	30.8
20 - 21	4	25.0	10	38.4
22 - 23	3	18.8	5	19.2
24 - 25	1	6.3	1	3.8
over 25			2	7.6

Table 2 presents the group values for the PEPS

variables. For an individual, a variable value between 41 and 59 indicates no special needs for that variable.

Table 2

PEPS Variables by Group

	Control (n=16)		Experimental (n=26)	
	M	SD	M	SD
Sound	55.60	4.29	53.04	6.55
Light	51.87	6.65	51.21	7.93
Warmth	48.53	10.03	46.63	7.82
Formal/Informal	43.20	10.92	46.38	7.86
Motivated/Unmotivated	52.53	5.41	50.00	5.91
Persistent	51.27	6.84	53.75	7.38
Responsible(conforming)	47.60	10.37	44.88	11.21
Structure	60.13	5.30	59.67	6.16
Alone/Peer	58.60	9.93	57.17	10.06
Authority	59.13	6.67	56.13	8.54
Several Ways	43.13	8.24	44.96	7.68
Auditory	54.80	12.20	56.25	8.47
Visual	49.00	8.46	50.50	9.31
Tactile	57.33	5.74	53.63	6.52
Kinesthetic	53.60	6.45	52.88	4.85
Requires Intake	59.40	7.47	56.08	7.69
Evening/Morning	43.93	5.57	45.33	6.40
Late Morning	40.87	10.74	46.42	6.94
Afternoon	63.53	8.97	60.75	9.06
Mobility	56.87	7.31	54.25	6.82

Table 3 presents the means and standard deviations of the pre-treatment measures for the two groups on the variables. High school math GPA was calculated using the actual averages from the transcripts and using a 4.0 scale. ACT Math and ACT English scores were obtained from either the high school or college transcript. The Watson-Glaser Critical Thinking Appraisal score is the raw score. The maximum possible score on the WGCTA is 80. The means for the two groups are quite close on all measures. The control group means are slightly higher on high school math GPA, ACT math score and WGCTA.

Table 3

Pretreatment Measures by Group

	Control (n=16)		Experimental (n=26)	
	M	SD	M	SD
High School Math GPA	3.2415	.6655	3.1517	.8347
ACT Math Score	21.9231	4.4993	19.0833	3.4756
ACT English Score	19.0000	5.8023	19.9167	4.2212
WGCTA	51.2800	9.6000	49.2900	10.0200

Table 4 presents the post-treatment variables by group. The college GPA and the college math GPA are the cumulative grade point averages calculated on a 4.0 scale at the end of the semester in which the treatment occurred. The final

examination had 224 possible points which were then coded for this study as a percentage on a 100 point scale. Final course average is also a percentage on a 100 point scale. The experimental group means for college GPA, college math GPA, and final examination score are higher than the control group. These means are descriptive and were not calculated using the controlling factors.

Table 4

Post-treatment Measures by Group

	Control (n=16)		Experimental (n=26)	
	M	SD	M	SD
College GPA	2.6219	.8426	2.8396	.5689
College Math GPA	1.9413	1.3331	2.5769	1.1559
Final Exam Grade	57.1425	24.0251	57.6219	26.9568
Final Course Average	71.3788	19.4262	70.6954	20.8212

Test of Hypotheses

In this section, the hypotheses will be restated along with decisions of acceptance or rejection based on the data used to test the hypothesis.

This study used the non-equivalent control group design because the subjects could not be randomly assigned to the treatment groups. The choice of group to be used for the treatments was randomly assigned. An analysis of covariance

was used as the statistical test. Significance was decided at an alpha level of .05 on the tests of the hypotheses.

Hypothesis 1

First semester programming students whose programming assignments are concerned with writing specialized modules that are part of well developed pre-written programs will score differently on the final examination from students who write programs from scratch for each assignment during the semester while controlling for learning style, critical thinking skills, ACT mathematics score, and ACT English score.

Hypothesis 1 was accepted. Table 5 shows that there was a significant effect for group on the variable of final examination grade while controlling for ACT math, ACT English, WGCTA, and the PEPS variables persistent, visual preferences and tactile preferences(F (1,26) = 6.306; p = .019). Students who added to or modified well-written, well-documented programs as assignments scored higher on the final examination for the course than students who wrote programs from scratch. The mean on the final examination adjusted for factors and covariates was 55.8033 for the control group and 62.0232 for the experimental group. The predicted mean adjusted for factors and covariates was 47.7893 for the control group and 66.3944 for the experimental group.

Table 5

Test of Hypothesis 1: Differences between Groups on the Variable of Final Exam

	ANCOVA Summary Table				
	SS	df	MS	F	p
Covariates					
Combined	9388.147	6	1564.691	5.902	.001
ACT Math	133.434	1	133.434	.503	.484
ACT English	1182.012	1	1182.012	4.458	.044
WGCTA	1827.624	1	1827.624	6.893	.014
Persistent	544.967	1	544.967	2.056	.164
Visual	779.517	1	779.517	2.940	.098
Tactile	1519.134	1	1519.134	5.362	.010
Main Effects					
For Group	1695.181	1	1695.181	6.306	.019
Residual	6893.253	26	265.125		
Total	17953.218	33	544.037		

Hypothesis 2

The average final semester grade for the two groups will be different while controlling for learning style, critical thinking skills, ACT mathematics score, and ACT English score.

Hypothesis 2 was accepted. Table 6 shows that there was a significant effect for group on the variable of course average while controlling for ACT math, ACT English, WGCTA,

and the PEPS variables persistent, visual preferences and tactile preferences (F (1,26) = 4.536; p = .043). The mean for course average adjusted for factors and covariates was 69.8892 for the control group and 73.4523 for the experimental group. The predicted mean adjusted for factors and covariates was 63.9563 for the control group and 76.6884 for the experimental group.

Table 6

Test of Hypothesis 2: Differences Between Groups on the Variable of Course Average

ANCOVA Summary Table

	SS	df	MS	F	p
Covariates					
Combined	6752.613	6	1125.436	6.520	.000
ACT Math	149.087	1	149.087	.864	.361
ACT English	999.492	1	999.492	5.791	.024
WGCTA	1200.265	1	1200.265	6.954	.014
Persistent	521.839	1	521.839	3.023	.094
Visual	256.388	1	256.388	1.485	.234
Tactile	1581.265	1	1581.265	9.161	.006
Main Effects					
For Group	782.930	1	782.930	4.536	.043
Residual	4487.661	26	172.602		
Total	12023.205	33	364.340		

Qualitative Discussion and Observations

Both the control group and the experimental group were given identical assignments for the first two programs. The differences in the assignments thereafter were that while the control group wrote their programs from scratch, the experimental group was given a program template that contained a skeleton program consisting of program elements that had already been covered and elements that were yet to be covered.

The students in the experimental group struggled with the program templates. While they seemed to understand how to write the parts of the program that they were responsible for in isolation, they had to work very hard to understand the workings of the template programs that they were adding to even though the templates included very detailed descriptions of each unfamiliar control structure in the internal documentation.

The initial template (assignment 3) was written in such a way that it was possible to compile, run, and test the program before the students added anything to it. At run-time, the program displayed a menu and requested that the user choose 1 to run a temperature conversion routine, 2 to calculate the distance to a rainbow, 3 to perform a decimal to binary conversion, or 9 to stop execution. The

temperature conversion routine was a fully functional interactive program meant to be an example for the assignment. The second and third choices were function stubs that displayed a message stating that the routine had not yet been implemented. One of these messages was to be replaced with the student implementation of the programming problem. The students had been instructed to read through the printed copy of the source code and then attempt to follow the order of execution while they ran the program choosing each of the menu items. The template had been developed to illustrate concepts that were not discussed in the lecture until several weeks later, but each was described in detail in the program documentation. A flag-controlled while loop allowed the menu to be displayed repeatedly until the student chose to quit. Each menu item choice resulted in the execution of a different function module and was meant to illustrate functional decomposition.

Each assignment template was more complex than the previous ones and required more time for the students to understand before they began their modifications. This struggle was such that the researcher / instructor was convinced that the experimental group subjects would in fact perform poorly on the final examination when compared to the control group.

The use of a carefully planned function interface

including preconditions and postconditions was discussed in both groups, but the experimental group had been reading this kind of documentation in the program templates for several weeks before they were asked to design it themselves. The function interface documentation written by the experimental group showed a far better understanding of the concept.

The control group had to understand only the programming problem itself to begin each assignment. They voiced far fewer complaints about the difficulty of the assignments in general. Student evaluations from the fall semester did not show a marked difference in attitudes of the two groups towards the course or the instructor.

The PEPS variable data is presented below in crosstabulated format. Scores on any variable that fall between 41 and 59 indicate that the subject has no special environmental needs where that variable is concerned. The instructor's scores are indicated by an asterisk and bold face in the left column. Below each table is a short description of the recommendations for the scores outside of the 41-59 range with the actual classroom or course conditions appearing underlined. Table 7 presents the "immediate environmental variables" of sound, light, warmth, and formal/informal design.

Table 7

Immediate Environmental Variables by Group

		Control		Experimental	
		count	%	count	%
Variable		*n*=15		*n*=24	
Sound	< = 40			1	4.2
*54	41-59	11	73.3	18	75.0
	> = 60	4	26.7	5	20.8
Light	< = 40	2	13.3	4	16.7
	41-59	11	73.3	18	75.0
*61	> = 60	2	13.3	2	8.3
Warmth *35	< = 40	1	6.7	4	16.7
	41-59	12	80.0	19	79.2
	> = 60	2	13.3	1	4.2
Formal/Informal Design	< = 40	4	26.7	4	16.7
	41-59	11	73.3	19	79.2
*64	> = 60			1	4.2

Sound: Score of 40 or below: establish silent work areas, alcoves. Score of 60 or above: provide soft music, conversation areas, open office environment.
Light: Score of 40 or below: work spaces with indirect light. Score of 60 or above: windows, lamps, bright light
Warmth: Score of 40 or below: air-conditioning, ventilation, cooler areas, cool colors. Score of 60 or above: warmth, enclosures, screens, warm colors

Formal / Informal Design: Score of 40 or below: informal climate - soft chairs, pillows indirect lighting. Score of 60 or above: rows of desks, direct lighting.

Table 8

Emotionality Variables by Group

		Control		Experimental	
		count	%	count	%
Variables		n=15		n=24	
Motivated / Unmotivated	< = 40			1	4.2
	41-59	13	86.7	20	83.3
*60	> = 60	2	13.3	3	12.5
Persistent	< = 40	1	6.7		
*56	41-59	13	86.7	19	79.2
	> = 60	1	6.7	5	20.8
Responsible (conforming)	< = 40	2	13.3	8	33.3
	41-59	11	73.3	14	58.3
*60	> = 60	2	13.3	2	8.3
Structure	< = 40				
	41-59	6	40.0	10	41.7
*60	> = 60	9	60.0	14	58.3

Motivated / Unmotivated: Score of 40 or below: short-term, simple, assignments requiring frequent discussions with instructor. Solicit self-developed goals. Score of 60 or above: encourage self designed objectives before instructor assesses effort. Permit self pacing and rapid achievement.

Persistent: Score of 40 or below: short-term assignments, options based on individual interest. Permit attention to multiple tasks simultaneously. Praise during process. Score of 60 or above: long-term assignments; supervision only when necessary. Praise at completion.

Responsible conforming: Score of 40 or below: short-term assignments with single or dual goals; provide options. Simple directions. Interest based. Praise and reward throughout. Explain import of project. Score of 60 or above: gradually increase length and scope of assignments. Challenge at or slightly beyond functional ability.

Structure: Score of 40 or below: clearly stated objectives. Choice of resources, procedures, time lines, environmental, sociological and physical elements. Provide creative options - review work often. Score of 60 or above: Precise instructions on all aspects. Leave nothing to interpretation. Gradually lengthen and allow choices. Specific working and reporting patterns and criteria.

Table 9

Sociological Variables by Group

		Control		Experimental	
		count	%	count	%
Variables		n=15		n=24	
Alone/Peer Oriented	< = 40			2	8.3
Learner *59	41-59	9	60.0	12	50.0
	> = 60	6	40.0	10	47.7
Authority Oriented	< = 40				
Learner	41-59	6	40.0	13	54.2
*60	> = 60	9	60.0	11	45.8
Learns in Several Ways	< = 40	6	40.0	9	37.5
*59	41-59	8	53.3	15	62.5
	> = 60	1	6.7		

Alone / Peer Oriented: Score of 40 or below: self-designed objectives, procedures and evaluations before instructor assesses. Self pacing and achievement beyond goals. Encourage creativity. Score of 60 or above: pair or team with others with the same sociological characteristics. Encourage meetings. Allow team to evaluate each other. Authority Oriented: Score of 40 or below: permit isolated achievements self-oriented, worker groupings is peer oriented or multiple options if learning in several ways is indicated. Score of 60 or above: place near instructors, schedule frequent meetings. Check work often. Provide

frequent feedback.

Several Ways: Score of 40 or below: permit work in preferred sociological pattern. Recheck self-orientation and motivation, responsibility and persistence. Score of 60 or above: provide a variety of working patterns; alone, grouped with peers and supervisors; use varied resources.

Table 10

Physical Needs Variables by Group

		Control		Experimental	
		count	%	count	%
Variable		n=15		n=26	
Auditory	< = 40	2	13.3	1	4.2
	41-59	8	53.3	14	58.3
*67	> = 60	5	33.3	9	37.5
Visual	< = 40	3	20.0	4	16.7
*59	41-59	11	73.3	16	66.7
	> = 60	1	6.7	4	16.7
Tactile	< = 40	-	-	-	-
	41-59	8	53.3	19	79.2
*66	> = 60	7	46.7	5	20.8
Kinesthetic	< = 40	-	-	-	-
	41-59	12	80.0	21	87.5
*63	> = 60	3	20.0	3	12.5
Requires Intake	< = 40	-	-	-	-
	41-59	7	46.7	15	62.5

*67	> = 60	8	53.3	9	37.5
Evening / Morning	< = 40	4	26.7	5	20.8
	41-59	11	73.3	19	79.2
*64	> = 60	-	-	-	-
Late Morning	< = 40	7	46.7	4	16.7
*55	41-59	8	53.3	20	83.3
	> = 60	-	-	-	-
Afternoon *38	< = 40	-	-	-	-
	41-59	6	40.0	14	58.3
	> = 60	9	60.0	10	41.7
Mobility	< = 40	-	-	-	-
*43	41-59	10	66.7	19	79.2
	> = 60	5	33.3	5	20.8

Auditory: Score of 40 or below: use resources indicated under perceptual preferences. If none are 60 or more, use multi-sensory resources: computers, videotapes, tactile/kinesthetic materials. Suggest reading and note-taking prior to lecture. Score of 60 or above: use tapes, videotapes, records, radio, television and precise oral directions.

Visual: Score of 40 or below: use resources indicated under perceptual preferences. If none are 60 or more, use multi-sensory resources: computers, videotapes, tactile/

kinesthetic materials. Suggest lecture and note-taking prior to reading. Score of 60 or above: pictures, filmstrips, computers, films, graphs. Should read prior to lecture.

Tactile: Score of 40 or below: use resources indicated under perceptual preferences if strong. If none are 60 or more, use multi-sensory resources: computers, videotapes, tactile/kinesthetic materials. Note-taking and manipulatives will be less effective than readings and lectures. Score of 60 or above: use manipulatives and 3-D materials. Allow these individuals to plan, demonstrate, report and evaluate using models and real objects. Encourage them to keep written records.

Kinesthetic: Score of 40 or below: use resources indicated under perceptual preferences if strong. If none are 60 or more, use multi-sensory resources: computers, videotapes, tactile/kinesthetic materials. Score of 60 or above: provide opportunities for real and active experiences. Seeing projects in action and becoming physically involved are appropriate activities.

Requires Intake: Score of 40 or below: no special arrangements. Score of 60 or above: arrange frequent opportunities for food breaks, allow beverages at desk and so on.

Evening / Morning: Score of 40 or below: schedule difficult

tasks in evening. Flex-time self scheduling. Score of 60 or above: schedule difficult tasks in the morning. (Please note experimental group was an 8:00 am class and no student scored in this range, but the instructor did)

Late Morning: Score of 40 or below: schedule difficult tasks in strongest segment of the time energy curve. Score of 60 or above: schedule difficult tasks in late morning.

Afternoon: Score of 40 or below: schedule difficult tasks in strongest segment of the time energy curve. Score of 60 or above: schedule difficult tasks in the afternoon.

Needs Mobility: Score of 40 or below: stationary desk or workstation where most responsibilities can be completed. Score of 60 or above: provide frequent breaks, assignments that require movements to different locations and schedules that build mobility into the learning pattern.

Observations Beyond The Study Period

During the spring semester, the researcher / instructor taught a single section of Introduction to Programming II, the second course in the sequence. The students were almost evenly split between control group participants and experimental group participants from the fall semester. In this course all programming assignments were written from scratch and the emphasis is on writing efficient programs consisting of tightly designed modules and programmer-designed data types.

The programs written by the experimental group participants were designed using far better examples of functional decomposition than those from the control group. The experimental group participants showed a firmer grasp of the concepts of formal versus actual parameters in function calls, function declarations, function definitions, and the use of global versus local variables. Documentation in the experimental group participant programs was more extensive than that of the control group participants.

It is also worth noting, that the experimental group participants incorporated design features such as error traps and menus that they had seen in the previous semester's program templates into the programs that they wrote for the programming II course. While the students may not have understood these structures when they experienced them initially, they seemed to see how these structures improved the overall look and feel of a finished project.

CHAPTER V

SUMMARY, CONCLUSIONS, DISCUSSION, LIMITATIONS, AND RECOMMENDATIONS

Summary

The purpose of this study was to compare student achievement in Introduction to Programming I between students who were given a series of well-developed programs which they added to or modified and students who were required to write a series of entire programs while controlling for learning style, score on the WGCTA, ACT math, ACT English, and high school mathematics GPA.

Summary of Procedures

Forty-two students attending Pearl River Community College during the 1998-1999 academic year and enrolled in Introduction to Programming I were the subjects for this study. Three sections of the course were offered. Treatment was randomly assigned to one section in the fall term. The single spring section was used to increase the number of students in the experimental group.

The pre-treatment measure of critical thinking ability was determined by administering the Watson-Glaser Critical

Thinking Appraisal to each subject. Student learning style was determined by administering the Productivity Environmental Preference Survey. High School and college transcripts provided information on the grade point averages for high school mathematics, college mathematics and overall college course work as well as ACT mathematics score and ACT English score. Student achievement in Introduction to Programming I was determined using the final examination grade and the final course average which included the scores for all tests and programming assignments.

Data collected from these sources was used to determine if there were differences in achievement in the programming course between those students who added to well-written, well-documented programs and those who developed and wrote complete programs as assignments.

Summary of Major Findings

Results of the analysis of data may be summarized as follows:

1. Hypothesis 1: There was a significant group effect for final examination grade while controlling for ACT mathematics score, ACT English score, critical thinking skill, and learning style. The Hypothesis was supported by the data. After controlling for the covariates, the experimental group had a higher mean final examination score

than the control group.

2. Hypothesis 2: There was a significant group effect for course average while controlling for ACT mathematics score, ACT English score, critical thinking skill, and learning style. The Hypothesis was supported by the data. After controlling for the covariates, the experimental group had a higher mean course average than the control group.

Discussion

Computer science education as a discipline is quite young and is continuing to emerge as a recognized entity. In view of the fact that computer science courses, as with most scientific / analytically based disciplines, builds on the knowledge acquired from previous courses, the research would support the need to examine the effectiveness of teaching methodology for the introductory courses. A firm foundation in the fundamental material should lead to continued success in computer science. For those students who take this introductory course as part of the core requirements for other disciplines, it is imperative to make this course as effective as possible since it is, in many cases, the only exposure to programming that they receive in an academic setting.

In this study population, students who were asked to modify or add to existing well-developed well-documented programs as programming assignments, performed better on the final examination and had higher course averages than students who wrote programs from scratch while controlling for pretreatment factors. In general, this result appears to be consistent with the findings of cognitive psychologists (Perfetti, 1979, Miller, 1956) and proponents of the use of advance organizers (Ausubel, 1960, 1968, 1978, Mayer, 1979) that force a deeper processing on the part of the learner.

A learning style directly impacts successful methods of learning that occur both inside and outside of the classroom setting. The program templates were developed to act as advance organizers for material that was covered later in the lecture portion of the course. In order to add their code to the templates the students were forced to read the documentation that explained the pre-written modules of code and attempt to understand their function within the program as a whole. This study activity occurred outside of the classroom. While the learning activities and environment inside of the classroom were carefully controlled to be equivalent for both groups, the only activity carried out outside of the classroom that can be reasonably assumed to be categorically different is that the experimental group

had to read the templates and make some sense of them before writing their assignments. Had the students been urged to use the PEPS findings to direct their study environments, it is possible that even greater benefit could have been realized.

The program templates were developed to act as advance organizers for material that was covered later in the lecture portion of the course. In order to add their code to the templates the students were forced to read the documentation that explained the pre-written modules of code and attempt to understand their function within the program. This study activity occurred outside of the classroom. A learning style directly impacts successful methods of learning that occur both inside and outside of the classroom setting.

Second language acquisition (SLA) research has many conflicting findings about the characteristics of good second language learners. There are, however, some findings that are relative to this study. Learning styles do play a part in the acquisition of a second language. Visual learners need to see the language construct under investigation and aural learners need to hear it (Lightbrown & Spada, 1995). Since visual preferences played a part in the outcomes of this study, it is reasonable to conclude that the code examples given to the experimental group as

program templates facilitated a deeper understanding of those concepts. By allowing the students to see their additions and modifications as integral parts of a larger whole, their perspective is changed. Perhaps the relative import of the placement of a single statement or small block of statements is realized more fully when those statements are integrated into a larger program than when those statements only impact each other as they would in a smaller, more focused program.

Another parallel between the template methodology used in this study and second language acquisition can be drawn from the discussions on effective second language acquisition in natural settings versus instructional settings. Lightbrown tells us that in natural language settings the learner is "exposed to a wide variety of vocabulary and structures" (p. 71). In these settings they generally use their limited knowledge of the second language in brief encounters such as greetings, purchasing transactions, and information exchanges. In one-on-one conversations modifications to incorrect language usage are occasionally offered. The template programs to some extent provide the wide variety of language structures in complex settings that a second language learner would encounter in a foreign language environment. The brief encounters can be thought of as the modifications and small additions made to

the templates. The compiler, to some extent, provides the language correction that is usually gained in one-on-one conversation in the natural setting in that the compiler points out the syntax (grammar) errors and the surrounding code, then provides clues to the possible correction of the error.

The less structured environment of one-on-one language learning is generally thought of as less stressful for the learner and in many ways more productive than a classroom environment. The compiler does not provide positive feedback in any real sense unless it is the gratification from having a program run successfully at some point. More importantly perhaps is the fact that it does not provide judgmental or negative feedback either. Krashen (1982) maintains that error correction in the classroom is counter productive in the acquisition process since it tends to make the student more cautious or afraid to respond to questions and generally puts the student in a defensive frame of mind (p. 76). Working directly with the computer, the student is free to 'play' with their program code without supervision much like the infant plays with words and sentence structure when they are alone and in the early stages of language acquisition and indeed as students do when writing and refining the various drafts that lead to a finished paper.

In the structured classroom, especially on test days,

the emphasis is generally on the syntactic correctness of isolated structures. Language drills teach the student how to reply to the somewhat predictable questions asked by the instructor. Success in terms of testing in classroom language learning settings does not generally lead to conversational fluency in the spoken language. This is analogous to computer science classrooms where there are probably many students who test (reply) well and program (converse) poorly.

Ellis (1985), with regard to SLA, tells us that errors in the way the learner produces statements in the target language provide important clues to the way the input data is incorporated into their cognitive structures. If analysis of the errors could show them to be systematic in some way, presumably some adjustment to the delivery of the input data would correct the misconception. Ellis points out that this would be quite difficult with natural languages due to the number of contexts in which the error potential exists. An example of this might be that subject-verb inversion occurs in what-where statements, but does not occur in who-when statements produced by a given learner or indeed an entire classroom full. The researcher suspects that systematic analysis of errors would be much easier with programming languages given that the numbers of words, statements, and contexts under which the errors could occur are much

smaller.

While not directly a part of this study, the researcher / instructor participated in an anecdotal study that concentrated on student errors in while-loop and for-loop construction several years ago. Algorithms for programming assignments that were to contain while-loops and for-loops were visually analyzed and error categories were developed for the mistakes in loop construction. Focused lectures were then carefully crafted around the error categories in an attempt to prevent the misconceptions surrounding those constructs and their uses, i.e., the input data was adjusted to resolve some of the misconception problems. Those adjustments to concept coverage have remained in use. Since comprehensible input is a prerequisite for language acquisition, similar techniques to pinpoint possible problem areas in content delivery could be incorporated into any level computer science course.

Limitations

As with any study involving human subjects, there were a number of variables outside of the control of the researcher/ instructor. First, the sample population came from a community college that serves a five county rural area in south Mississippi. Conclusions are therefore only generalizable to similar populations.

The study was originally planned to be carried out in a single semester. The original experimental group sample went from 28 subjects down to 10 in the first five weeks of the semester. Realizing that the sample lacked sufficient statistical power, the researcher went forward with the idea of turning the semester's work into a pilot study. The researcher / instructor continued the treatments and kept careful records of weekly course content. The opportunity to teach the single spring semester section allowed the experimental treatment to continue with this group. At the end of the spring semester, the two experimental sections were analyzed and the decision was made to collapse these groups into a single experimental group for the purposes of this study to ensure a strong study population for statistical purposes.

With a limited population, missing data can become a source of possible inconsistency in the findings. There were 2 subjects in each of the control group and the experimental group who missed the days that the pre-treatment measures were administered. One of the subjects in the experimental group and two subjects in the control group had not graduated from traditional secondary schools but had earned equivalency certification which meant that high school mathematics GPA was unavailable for those subjects. In addition one student in the control group and

four in the experimental group transferred credits from other post-secondary institutions making their high school transcripts unavailable.

Recommendations for Further Study

Based on the results of this study the following recommendations for further research are indicated:

1. Replication of this study at several universities and / or colleges would allow generalization to a broader population and might indicate the role of more factors in the process than were identified in this study. A larger, more diverse population might provide data to further validate this study.

2. Development of a longitudinal study following those group participants who are majoring in computer science through their degree programs would allow a more complete picture of the impact of these methods on later course work.

3. Similar studies should be developed for other higher level computer science courses that teach a language or paradigm that differs from the CS1/CS2 language and/or paradigm (i.e., C++ followed by Ada or functional followed by object oriented) taught at a given university. If the initial programming assignments were template style assignments, similar outcomes (i.e., better use of modularization, better overall programming form, increased

ability to maintain existing code) might be observed in these later courses as well.

Student Consent Statement

Periodically, Pearl River Community College conducts or participates in studies to ascertain the effectiveness of classroom teaching methods. This semester certain courses will be evaluated. This particular class is one of those that has been chosen to participate in such a study. It is hoped that you will agree to cooperate by consenting to participate. While it is not imperative that you participate, the format of the course will not change whether you participate or not. In other words, the study will be transparent to you as a student. It is, however, necessary to ask that you indicate that you do not object to such a process.

You should be aware that we feel very strongly about improving the quality of instruction. Your complete cooperation will not go unnoticed. ***Your grade will not be adversely affected by this study***. **Indeed it is hoped that the teaching methods employed will result in an improvement in your understanding of the concepts in this course.** We hope that it makes this course more enjoyable and easier to comprehend.

This project has been reviewed by the Human Subjects Protection Review Committee which ensures that research projects involving human subjects follow federal regulations. Any question or concerns about rights as a research subject should be directed to the Directory of Research and Sponsored Programs, University of Southern Mississippi, Box 5157, Hattiesburg, MS 39406, (601) 266-4119.

I understand that this course is part of a study and have no objections to participating.

Signed,

BEST COPY AVAILABLE

Program Grading Criteria

71

consistent formatting

Criteria Grades (0 to 5 points each):

Program Grade Sheet

APPENDIX B
THE EVALUATION OF PROGRAM QUALITY

BEST COPY AVAILABLE

APPENDIX C
STUDENT INFORMATION SURVEY

CSC 1613
Student Information Survey

1. Student Name : _____
2. Student ID : _____
3. Gender : _____
4. Age : _____
5. Race : _____
6. Major : _____
7. College GPA : _____
8. H.S. Math GPA : _____
9. Composite ACT : _____
10. Math ACT : _____
11. English ACT : _____
12. # of H.S. Math courses taken : _____
13. # of College Math courses taken : _____
14. Grade earned in college algebra : _____
15. Cumulative college math GPA : _____

APPENDIX D
CSC 1613 COMPREHENSIVE FINAL EXAM

FALL 1998 **INTRODUCTION TO PROGRAMMING I** **FINAL EXAM**

Name: _____ Grade: _____ / 224

1. Mark the following C++ identifiers as valid (V) or invalid (I).

_____	`big10`	_____	`the12DaysOfChristmas`
_____	`merryChristmas`	_____	`happy holidays`
_____	`12DaysOfChristmas`	_____	`happyHolidays!`

2. Given the three syntax templates below, mark each "Santa Claus" as valid (V) or invalid (I).

`SantaClaus`

```
Sleigh . . . Reindeer
```

`Sleigh`

`Reindeer`

_____	♫♫♫=	_____	♫+☆
_____	☆☺♫=/	_____	☺=
_____	+/=☺	_____	♫☆/

3. Indicate the type of each expression below: float (F), integer (I), or invalid (I), show the result, using the correct data type.

Type	Result		Type	Result	
_____	_____	9 - 4	_____	_____	9.0 / 4.0
_____	_____	9.0 + 4.0	_____	_____	9 / 4
_____	_____	9 % 4	_____	_____	4 / 9
_____	_____	9.0 % 4.0	_____	_____	

4. Given the following: int1 and int2 are int variables; floatVar is a float variable: `int1 = 2, int2 = 8,` and `floatVar = 3.7,` determine the type (I or F) of the result of each expression below.

Type		Result			Type		Result		
				`int1 / float(int2)`					`fabs(-63.2)`
				`2.0 / int1 + 7`					`sqrt(36.0)`
				`floatVar * int1`					`6 + int(4.9) * 3`
				`int1 + int 2 % 5`					`pow(3.0, 2.0)`
				`(1 /5) * int1`					`abs(int2 * (3))`
				`int(floatVar+0.5)`					`sqrt(sqrt(9.0) + fabs(-13.0))`

5. What is the output of the following program fragment?

```
holly = 56.789;
mistletoe = 21;
evergreen = 76543;
cout << endl << setw(12) << evergreen;
cout << mistletoe << endl;
cout << setw(8) << setprecision(1) <<holly << endl;
cout << endl << setw(14) << setprecision(6) << holly  << endl;
cout << '$' << setw(5) << setprecision(2) << holly << endl;
```

Position	0 1	0 2	0 3	0 4	0 5	0 6	0 7	0 8	0 9	1 0	1 1	1 2	1 3	1 4	1 5	1 6	1 7	1 8	1 9	2 0	2 1	2 2	2 3	2 4	2 5
Line 1																									
Line 2																									
Line 3																									
Line 4																									
Line 5																									
Line 6																									
Line 7																									
Line 8																									

6. Given that alpha, beta, and gamma are integer variables, and given these three lines of input data

```
999 888 777
666 555 555
444 333 222
```

what values are stored in alpha, beta and gamma after the following statements are executed?

```
cin >> alpha >> beta;
cin.ignore (10, '\n');
Cin >> gamma;
```

ANSWER: alpha _____ beta _____ gamma _____

7. Given these two lines of input data

```
$ X5
CDE
```

what values are stored in ch1, ch2 and ch3 by the following statements?

```
cin.get (ch1);
cin.get (ch2);
cin.get (ch3);
```

ANSWER: ch1 _____ ch2 _____ ch3 _____

8. Given this line of input data, what values are stored in int1,int2 and inchar by the following statement?

```
87.654 * 321
cm » int1 » inChar » int2;
```

ANSWER: int1 _____ inChar _____ int2 _____

9. Evaluate the following expressions as TRUE or FALSE.

___ `6 < 9 || 5 > 3 && 1 > 7`

___ `(6 < 9 || 5 > 3) && 1 > 7`

___ `8 > 2 && 0==0 || 1 < 3 && 5==6`

___ `!(8 > 2 && 0==0)|| 1 < 3 && 5==6`

10. Place an SC by each expression that will be evaluated by C++ with short-circuit evaluation and FE by each that will be fully evaluated:

_____ `5 == 8 && 8 > 4`
_____ `7 <= 10 && 5 >= 6`
_____ `4 < 9 || 3 <= 11`
_____ `5 == 3 || 8 >= 6`

11. What is the output of the following code fragment if the input value is 100?

OUTPUT:

```
cin  >> someInt;
if (someInt <= 50)
   cout << "Have a ";
cout << "Merry ";
cout << "Christmas!";
```

12. What is the output of the following code fragment? (All variables are of type `int`.)

```
alpha = 3;
beta = 20;
if (beta > 10)
 {
   int alpha = 5;
   beta = beta + alpha;
   cout << alpha <<' ' << beta << endl;
 }
cout << alpha <<' ' << beta << endl;
```

OUTPUT:

13. What is the output of the following code fragment? (All variables are of type `int`.)

OUTPUT:

```
x = 4;
y = 5;
z = 6;
Try (y, x);
cout << x <<' ' << y << ' ' << z << endl;

void Try ( int& a, int b )
{
  int x;
  x = a + 2;
  a = a * 3;
  b = x + a;
 }
```

14. Number the statements (with the blanks next to them) in the program below in the order they will be executed. (The line numbers are provided for use in question 16.)

```
1            #include <iostream.h>
2            void Change ( int, int&);
3            int a;
4            int b;
5            int main()
6            {
7   _____      a = 7;
8   _____      b = 10;
9   _____      Change (a, b);
10  _____      cout << "In function main,  the variables equal
                    << a << " and " << b  << endl;
11  _____      return 0;
             }

12           void Change ( int  x,
13                         int& y
             {
14            int b;
15  _____     b = x;
16  _____     y = y + b;
17  _____     x = y
18  _____     cout << "In function Change, the variables equal
                   << x << ", " <<  y << ", and " << b << endl;
             }
```

15. What is the output of the preceding program?

16. List all occurrences of these items from the program above using the line numbers provided:

local variable(s): _____

global variable(s): _____

formal parameter(s): _____

actual parameter(s): _____

reference parameter(s): _____

value parameter(s): _____

function prototype(s): _____

function declaration(s): _____

function call(s): _____

17. What is the output of the following code fragment? (Using the correct data type for the output is an important part of your answer.)

```
cout << Mystery ( 4.2 ) << endl;

int Mystery ( /* in */ float someval);
{
 if (someval > 2.0)
   return 3 * int(someval);
 else
   return 0;
}
```

OUTPUT:

18. This question demonstrates the hazards of side effects. What is the output of the following code fragment? (All variables are of type int.)

```
number = 4;
result = SumInts ( number );
cout << "The sum of the integers 1 through " << number
     << " is " << result << endl;

//*********************************
int SumInts (int& n)
{
 int sum = 0;
 while (n >=1)
  {
    sum = sum + n;
    n = n - 1;
  }
 return sum;
 }
```

OUTPUT:

Read through the list of terms on the last page of the exam. Use the terms to fill in the blanks below. Not all terms are used. No term is used more than once. Cross out terms as you use them. For best results try to do this exercise in two passes. Answer only the ones that you are absolutely sure of on the first pass. Let the process of elimination and your knowledge work for you. Feel free to tear off the list of terms page so that you can use it for all pages easily.

19. _____ a function declaration that includes the body of the function

20. _____ a specific set of data values along with a set of operations of those values

21. _____ the explicit conversion of a value from one data type to another.

    Example: `someInt = int(12.8);`

22. _____ the implicit (automatic) conversion of a value from one data type to another.
    Example: `someInt = 12.8;`

23. _____ general term for a control structure used when we want the computer to choose between alternative actions

24. _____ the execution of a series of statements (instructions) one after another, in order

25. _____ the repetition of statements (instructions) while certain conditions are met

26. _____ a program that translates a high-level language program into code that the computer can process

27. _____ the language made up of binary-coded instructions that are used directly by the computer

28. _____ a set of rules, symbols and special words used to construct a computer program

29. _____ a step-by-step procedure for solving a problem in a finite amount of time

30. _____ modifications are made to an existing computer program during this phase of the program's life cycle

31. _____ a general solution, or algorithm, is written during this phase of a computer program's life cycle

32. _____ coding of an algorithm takes place during this phase of the program's life cycle

33. _____ an individual pass through, or repetition of, the body of a loop

34. _____ the precedence that a local identifier in a function has over a global identifier with the same name in any references that the function makes to that identifier

35. _____ the condition that causes a loop to be exited

36. _____ the moment that repetition of the loop body ends and control passes to the first statement following the loop

37. _____ the moment the flow of control reaches the first statement inside the loop

38. _____ an input statement before the While loop, giving an initial value to the variable(s) that the loop will process

39. _____ a variable declared in a function heading

40. _____ a variable or expression listed in a call to a function

41. _____ an assertion that should be true after a module has executed

42. _____ a statement that associates an identifier with a data object, a function, or a data type so that the programmer can refer to that item by name

43. _____ a variable which is declared within a function and is not accessible outside the function

44. _____ an assertion that must be true before a module begins executing

45. _____ the memory address of the first element of an array

46. _____ What type of formal parameter is beta?

```
Void DoSomething (char& alpha,
                  char  beta);
```

47. _____ What type of formal parameter is alpha?

```
Void Dosomething (char& alpha,
                   char  beta);
```

48. _____ program code containing meaningful identifiers as well as judiciously used clarifying comments

49. _____ the C++ statement which means "Exit the entire loop immediately."

50. _____ the general name for a loop in which the loop test is positioned before the loop body

51. _____ a C++ looping structure whose body will always be executed at least once

52. _____ the order in which the computer executes statements in a progam

53. _____ a C++ looping structure that includes the initialization, testing, and incrementation of the loop control variable in a single location

54. _____ a measure of the effort expended by the computer in performing a computation

55. _____ a technique for developing a program in which the solution is expressed in terms of objects - self-contained entities composed of data and operations on that data

56. _____ a C++ control structure designed as an alternative to the IF THEN ELSE for certain multi-way branching tasks

57. _____ a location in memory, referenced by an identifier, in which a data value that can be changed is stored

58. _____ the region of program code where it is legal to reference that identifier

59. _____ the period of time during program execution when the variable has memory allocated to it

60. _____ a variable for which memory is allocated and deallocated when control enters and exits the block in which it is declared

61. _____ a variable for which memory remains allocated throughout the execution of the entire program

62. _____ an effect of one function on another that is not a part of the explicitly defined interface between them

63. Using a FOR loop, write the code to print the following line of output:

```
10  9  8  7  6  5  4  3  2  1
```

64. Write a nested IF THEN ELSE structure that prints the appropriate message according to the following information. Do not use any unnecessary comparisons.

When the value of inventory is...	Print this message...
Under 6	Time to reorder
7-20	Depleted stock
21-100	Normal stock
Over 100	Surplus stock

65. Rewrite the following *if then else* structure as an equivalent *switch* structure.

```
if ((dayNum == 1)||(dayNum == 2)||(dayNum == 4)||(dayNum == 5))
   cout << "Weekday" << endl;
else if ((dayNum = 3)
   {
     cout << "Hump day.
     cout << "Keep on going."  << endl;
   }
else if (dayNum == 6) || (dayNum == 0)
   cout << "Weekend" << endl;
else
   cout << "Day number must be 0-6." << endl;
```

66. Write a **void function** named `PlaceInorder` which accepts two integer parameters `intOne` and `intTwo` and places them in order so that `intOne` returns the smaller value and `intTwo` returns the larger value to the calling module (remember the cups). Include data flow comments, preconditions and post-conditions.

67. Write a **value-returning function** named `MilesPerHour` which accepts the distance traveled and the time traveled and returns the rate of travel in miles per hour. For example, when the function receives a distance traveled of 120.0 miles and a time traveled of 2.0 hours, it returns a rate of 60.0 miles per hour. The function and its parameters should be of type `float`. Include data flow comments, preconditions and postconditions.

BEST COPY AVAILABLE

Terms for Matching Questions 19 - 62

actual parameter
algorithm
assembly language
automatic variable
base address
break
case
compiler
continue
data type
declaration
documentation
do . . while
encapsulation
flow of control
for
formal parameter
function prototype
function definition
function declaration
global variable
high-level language
implementation phase
interface
iteration
lifetime of a variable
local variable
loop entry
loop exit
looping
machine language
maintenance phase
name precedence
object-oriented design
postcondition
post-test
precondition
pretest
priming read
problem-solving phase
programming language
reference parameter
scope of an identifier
selection
self documenting code
sequence
side effect
static variable
switch
termination condition
type casting
type coercion
value parameter
variable
while
work

APPENDIX E
PROGRAMMING ASSIGNMENT SET

Programming Assignment 0

Due Date:
Point Value: 20 points
Objectives: use the Turbo C** editor
create a new file
open an existing file and save it under a new name
write an algorithm

1) Punch in and save a program assignment heading to be used for all program assignment submissions. The heading should consist of: the programming assignment number, your name, course number and section, instructor, due date, point value, a section for programming objectives and a section for the algorithm. An example is shown below yours need not look exactly like this, but all required elements must be present. This file should be named heading.cpp

```
//*****************************************************************
//    CSC1613              Section 01          Instructor - Applin
//    Programming Assignment -
//    Author -
//    Due Date -                                Point Value -
//*****************************************************************
//    Objectives:
//
//*****************************************************************
//    Problem statement:
//
//*****************************************************************
//   Algorithm:
//
//*****************************************************************
```

2) Do programming Warm-Up Exercise #4 p. 90. Do not type the comment section shown in the book - use your template from part 1. Compile and run the program as described in the handout. Save this as prg0.cpp. Print one copy of the source code to hand in.

Programming Assignment 1

Due Date:
Point Value: 20 points
Objectives: edit and save a source code file
use the C** statements cout<< and <<endl;
compile and run a C** program

Everyone's first C program is a program that outputs their own name or initials, generally in large block letters made up of a combination of the character in question and blanks as needed such as:

```
   A
  A A
 AAAAA
A     A
A     A
```

pretty dull stuff...

Needlepoint, cross-stitch, fancy-work or embroidery all gives us alphabets that are rows of stitches and blanks spaces. Some of the alphabets are quite ornate others are very plain. The block layout of the needlework alphabets makes them perfect for programming output because the exact spacing is part of the pattern. A selection of needlework alphabet patterns is posted in the lab.

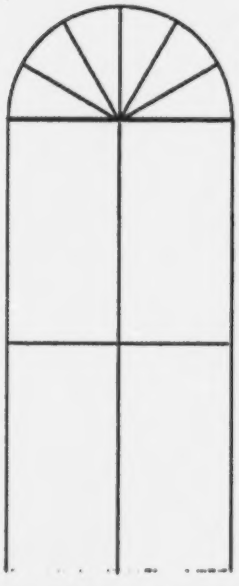
Write a complete C** program that outputs your name, initial(s), or a short message. Requirements will be fulfilled with a minimum of 3 letters in a simple pattern or a single Old English initial. You must keep in mind that the screen is 80 columns wide so your output must not exceed 80 characters. You need not write an algorithm for this programming problem.

Programming Assignment 2

Due Date:
Point Value: 20 points
Objectives: numeric calculation
use of named constants
develop a research strategy

No one is ever going to dump all of the facts that you need into your lap. You will always be required to look up things that you do not have. The question is where is the right place to look up what you don't have? Who can you ask? For this assignment your program documentation must include the name of the person that you **showed this assignment to**, what qualifications they have that prompted you to ask them, the name of the book that they gave you and the page number or numbers where you found the formulas needed to solve this problem. If you know where to look because of a previous course that is the source rather than a person.

An architect has been asked to design a dining room for a house in the Rocky Mountains. The dining room has two exterior walls and will be flanked by a porch that is accessible through strategically placed French doors. Between the doors there will be a series of windows designed to show a maximum of the spectacular view and incorporate a design reminiscent of the sun breaking the horizon. The arch is a perfect half-circle.



What follows is a description of the inside dimensions. We are not worried about the wood frames - just the glass. The length of the vertical from the floor to the point where the arch begins is 7.5 feet. The windows will be 36 inches wide. They do not open and are made of a **single piece of insulated double glass that includes the arch**. The price of each window is based on the total area, in square feet, of the glass that must be cut. This cost is $4.32/sq.ft.

Write a C** program that will calculate and output the total area of the piece of glass in each window and the cost of each window. Include appropriate user messages.

BEST COPY AVAILABLE

Programming Assignment 3

Due Date:
Point Value: 20 points
Objectives: use of the math.h library
named constants
formatted output of real numbers
interactive programming

How tall is a rainbow? Because of the way in which light is refracted by water droplets, the angle between the level of your eye and the top of the rainbow is always the same. If you know the distance to the rainbow, you can multiply it by the tangent of that angle to find the height of the rainbow. The magic angle is 42.3333333 degrees. The C++ standard library (and those of most other programming languages) works in radians and not degrees, so you will have to convert the angle to radians. The conversion formula is

radians = degrees *(π/180)

where π = 3.14159265

The header file math.h the C++ standard library provides a tangent function named tan. This is a value-returning function that takes a floating point parameter and returns a floating point result:

```
x=tan(someAngle);
```

If you multiply the tangent by the distance to the rainbow, you get the height of the rainbow.

Copy the file PRG3.CPP that you will find in the folder called **CSC 1613-02 Files** on the Public drive of the server. Print a copy of the program code. With the hard copy in hand, compile and run the program without making any changes. Try each menu item and attempt to enter something besides 1, 2, 3, or 9. Investigate the source code for the program features that you find new to you. Make sure that you understand why the program behaves as it does before proceeding with the assignment. You need not know how to write these features just gain an understanding of what is happening.

Modify the header included with that file so that it reflects your revision of the program.

Add the lines for formatted real number output in the area indicated in function main.

Write a top-down design and a C++ module to prompt the user for a single floating point value - the distance to the

rainbow - and compute the height of the rainbow. The module should print the distance to the rainbow and its height with phrases that identify which number is which. Display the floating point values to four decimal places. Add your code to the existing program by replacing the stub for. Format your module with consistent indentation, and use appropriate documentation as indicated.

Programming Assignment 4

Due Date: October 22, 1998
Point Value: 20 points
Objectives: formatted output
file output
nested if

The city of Mt. Pleasant bills its resident for sewage, water and sanitation services every 3 months (each quarter). The combined sewer and water charge is figured according to how much water is used by the resident. The scale is:

Amount (in gallons)	Rate
less than 1000	$0.03 per gallon
1000 - 1999	$25.0 + $0.02 for each gallon over 1000
2000 or more	$40.00 + $0.015 for each gallon over 2000

The sanitation charge is $4.50 per month. Write a top-down design and a C++ program that prompts the city secretary for the customer number (lot number) and the number of months for which that resident is being billed (1, 2, or 3. People can move in during a cycle.) and how much water was used. Write the output in the form of a statement to a file. The following is a sample bill:

Mt. Pleasant City Services

Residential Lot:	128
Months of Service :	2
Gallons of Water Used:	1200
Sewage and Water Charge:	29.00
Sanitation Charge:	9.00
Total Charges:	38.00

You will begin with a template program as you did with program 3 that you will find on the server. Modify only those areas that are indicated.

Use the following data. Print the output file when you are finished.

127	3 months	2356 gallons	129	2 months	1454 gallons
128	2 months	1202 gallons	130	1 months	938 gallons

Programming Assignment 5

Due Date:
Point Value: 25 points
Objectives: use cin.get for character processing
while Statement with end-of-file
If Statements with other boolean conditions
event counting
nesting control structures
Character manipulation!

The Caesar Cipher is named after Julius Caesar, who first used it is a simple encryption method by which the alphabetic characters are shifted some number to encrypt a message. Nothing is done to the spaces or the numbers or the punctuation.

On the Internet there is a Unix utility called ROT13 which allows files full of questionable material (obscene, poor taste, etc) to be encrypted so that the user has to know how to decrypt them to read them. ROT13 is the Caesar Cipher with a rotation of 13 letters. A becomes N, a becomes n, B becomes O, b becomes o, and so forth.

There was a special cipher used by the confederate army - you could look it up and use it. Or you may choose to devise a cipher of your own - Just include a description of how the cipher works.

An ASCII chart can be found in the appendix of the textbook. Using it you should be able to see that certain mathematical corrections will have to be made to keep your encryption routine within the boundaries of the printable characters.

Develop a top-down design and write a C** program that will read the complete contents of a file and output the number of words in the file, the number of sentences in the file and the average length of the sentences, and encrypts the file as it processes it.

For our purposes a word is defined as a group of characters without blanks. A sentence is defined as a group of words ending with a period. The average length will be defined as the number of words divided by the number of sentences.

Programming Assignment 6

Due Date:
Point Value: 20 points
Objectives: Void functions
Value returning functions
code reuse
functional cohesion

What day were you born on and how many days have you been alive?

To calculate elapsed days we need a common way to measure days. Julian dates are consecutive numbers that begin in at 1 for January 1st and continue throughout the year. To do date arithmetic we can use a Julian date that is even more accurate by beginning the count on January 1st, 4713 B.C. The numbers for current Julian dates are above 2 million. To answer the questions above, we need to convert your birth date to a Julian date, find the day of the week, find the Julian date for today, and calculate the number of days that have elapsed since your birth. So how do we convert a Gregorian date to a Julian date?

Step 1. Given a year (an integer such as 1942), month (an integer between 1 and 12) and a day (an integer between 1 and 31), if the month is 1 or 2, then subtract 1 from year and add 12 to month.

Step 2. if the date is from the Gregorian calendar (after 10/15/1582) then compute an intermediate result with the following formula (otherwise let intRes1 equal 0):
intRes1 = 2-year/100+year/400 (integer division)

Step 3. Compute a second intermediate result using the formula:
intRes2 = int(365.25*year)

Step 4. Compute a third intermediate result using the formula:
intRes3 = int(30.6001 * (month + 1))

Step 5. Finally, the Julian date is calculated with the following formula:
JulianDate = intRes1 + intRes2 + intRes3 + day + 1720994.5

Your program should request today's date and a birth date in the form: mm dd yyyy. The month and day parameters must be within the prescribed ranges - make the user give you the correct input as discussed in class. We won't worry about the year so much. The program should convert them to Julian dates, display the day of the week the user was born and the

day of week for today then display how many days have elapsed between the two! The program should be fully modular with each module doing one specific task. GregToJul for example would convert any Gregorian date to a Julian one. DayOfWeek would calculate a day of the week based on a Julian date. GetDay, GetMonth, GetYear would each request and return a part of the needed Gregorian dates. Each should validate the input!

A template program is on the server. Read it carefully before you begin modification.

Note** The formulas above require nine significant digits. You must use double or long double for correct calculation.

Programming Assignment 7

Due Date:
Point Value: 25 points
Objectives: Programer Written Functions
For Loop
More on Boolean data

Goldbach's Conjecture

Goldbach's conjecture states that all even integers are the sum of two primes. This can be easily tested up to the computing limits of a given computer, but can not be proven above that. This is why it is only a conjecture and not a mathematical truth. It can not be proven, however it works for all numbers that have ever been tested.

Write a top-down design and a C++ program that:

- asks the user for an even integer. The program should test the input and force the user to give correct input.
- finds and reports the two primes that sum to that integer (while (!found))
- Continues this process until the user wishes to exit (while (!done) or do...while (!done))

Your design should contain a Boolean value returning function called IsPrime that receives a single integer value and tests it. ***Boolean IsPrime(int number)*** The call from function main should be something like this:

```
found = IsPrime(half1) && IsPrime (half2);
```

The design should be fully modular. I suggest the following (these are suggestions only)

```
void Continue (Boolean& done)
void GetAnInteger (int& number)
```

APPENDIX E
SAMPLE PROGRAM TEMPLATE

```
//*******************************************************************************
*
//    CSC1613                 Section              Instructor - Applin
//    Programming Assignment -  03
//    Author                  -  Anne Applin
//    Revised by              -
//    Due Date                -                    Point Value -
//*******************************************************************************
*
//    Objectives:
//*******************************************************************************
*
//    Problem statement:
//*******************************************************************************
*
//    Algorithm:
//*******************************************************************************
*

#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <float.h>

typedef int Boolean;          // this is a programmer defined data type

// All constant declarations go here
const Boolean TRUE = 1;
const Boolean FALSE = 0;
const char SPACE[14] = "             "; //array used for centering
output


// function prototypes
void Rainbow();
void ConvertDegreesF_to_DegreesC();
void BaseConversion();

//*******************************************************************************
*
int main ()
{
  Boolean done = FALSE;
  char selection;     //using a character helps prevent a runtime error

  // set up floating point output here
```


BEST COPY AVAILABLE

```
  // a while loop allows a block of code to be repeatedly executed.
  // the user can execute one module several times and/or run several
  // of the modules from the menu below.  The variable "done" has been
  // initialized to FALSE  (we are not (!) done, we just started).  One
  // of the options below will change the value to TRUE when that
happens
  // the condition (!done) will become FALSE and execution will continue
  // with the lines below the while loop block.

  while (!done)
  {
    clrscr();   /* clears the screen of the computer
                   at start of each iteration*/

    // the following statement draws a menu on the screen
    cout << endl << endl << endl
         << SPACE << "************************************************\n"
         << SPACE << "*    Enter the number of your selection        *\n"
         << SPACE << "*    *********************************         *\n"
         << SPACE << "*1. Calculate the height of a tree             *\n"
         << SPACE << "*2. Convert from Farenheit to Celcius          *\n"
         << SPACE << "*3. Convert a decimal number to another base   *\n"
         << SPACE << "*9. Quit                                       *\n"
         << SPACE << "*                                              *\n"
         << SPACE << "************************************************\n"
         << endl << endl;

    //  This is a user prompt
    cout << SPACE << "            Selection? ->  ";

    cin  >> selection;
    // the switch statement is a multi-way decision structure. Execution
    // will continue with one of the statements below based on what
value
    // is stored to the memory location called selection.
    switch (selection)
     {
       case '1': Trees();
                 break;
       case '2': ConvertDegreesF_to_DegreesC();
                 break;
       case '3': BaseConversion();
                 break;
       case '9': done = TRUE;
                 break;
       default : cout << "Invalid input please try again";
     } // end switch statement
    // if 9 is chosen the loop stops-otherwise the requested
    // program runs and then return to the top of the while loop
  } // end while loop statement
  // For execution to continue here, the user has selected 9
  cout << SPACE << SPACE << "Have a nice day!!";
  return (1);
} // end function main
//*************************************************************************
//  This is called a stub.  Everything here is syntactically correct
```

```
//  so that the program will run even though this part hasn't been
// written yet.  Delete the first cout statement and add your code
there.
// Change these comments to reflect the purpose of this module.
void Rainbow()
{
  char anyKey;
  clrscr();       // clear the screen

  cout << endl << endl
       << SPACE << " Rainbow module not yet implemented \n";
  // do not change the lines below!
  cout << SPACE << "    - Press ENTER to continue";
  cin.ignore(1,'/n');
  cin.get(anyKey);
} // end of function Tree
//*****************************************************************************
*
//The following function accepts user input of a temperature and
converts
//it to the equivalent celcius temperature.  It then echos the user
input
//displays the converted value with appropriate messages for the user.
void ConvertDegreesF_to_DegreesC()
{
 float degreesC, degreesF;
 char anyKey;
 clrscr();       // clear the screen

 cout << endl << endl << endl;

 //  tell the user what is going on here and prompt for input

 cout << SPACE << "  This program converts a temperature \n"
            << SPACE << "from degrees Fahrenheit to degrees Celcius.\n"
            << SPACE << " Enter a temperature  ->  ";
 _fpreset(); //this resets the floating point process so that a new
value
            // can be read in without an error occurring (TURBO
specific)
 cin >> degreesF;

 // do the temperature conversion
 degreesC = (degreesF - 32.0) * 5.0 / 9.0 ;

 // output the result with appropriate labels
 cout << endl << endl << setprecision(2)
      << SPACE << degreesF << " degrees Farhenheit equals "
      << degreesC << " degrees Celcius.\n\n\n"
      << SPACE << "press ENTER to continue";

 // these lines pause the screen so that the user can read the result.
 // Without them the program redisplays the main menu before you have
 // time to see the conversion
  cin.ignore(1,'/n');
  cin.get(anyKey);
}   // end of function ConvertDegreesF_to_DegreesC
```


BEST COPY AVAILABLE

```
//*****************************************************************************
*
void BaseConversion()
{
  // this is another stub.
  char anyKey;
  clrscr();
  cout << endl << endl
       << SPACE << " Base Conversion not yet implemented "
       << endl << SPACE << "     - Press ENTER to continue";
  cin.ignore(1,'/n');
  cin.get(anyKey);

}  // end of function BaseConversion
```

```
//*********************************************************************
//    CSC1613                    Section 02         Instructor - Applin
//    Programming Assignment -   04
//    Author                 -   Anne Applin
//    Revised by             -
//    Due Date               -                      Point Value - 20
//*********************************************************************
//    Objectives: Formatted output of integers and real numbers
//                file output
//                nested if statement
//*********************************************************************
//    Problem statement:
//
//*********************************************************************
//    Algorithm:
//
//*********************************************************************

//*********************************************************************
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <float.h>
#include <ctype.h>
#include <fstream.h>

typedef int Boolean;             // this is a programmer defined data type

// constant declarations go here
const Boolean TRUE = 1;
const Boolean FALSE = 0;
const char SPACE[14] = "              "; // array used for centering output
const float BASE1_PER_G = .03;
const float BASE2 = 25.00;
const float BASE2_PER_G = .02;
const float BASE3 = 45.00;
const float BASE3_PER_G = .015;
const float SANITATION = 4.50;

// function prototypes
int GetResidentData(int);
void Error(int);
void PrintCharges(int);

// global variables ( things we want to use anywhere we need them)

ofstream invoiceFile;


//*********************************************************************
int main ()
{
  Boolean done = FALSE;
  char selection;     //using a character helps prevent a runtime error
  int customerNumber, digit;
  invoiceFile.open("invoices.dat");
  // set up floating point output here


  // a while loop allows a block of code to be repeatedly executed.
  // the user can execute one module several times and/or run several
  // of the modules from the menu below.  The variable "done" has been
  // initialized to FALSE  (we are not (!) done, we just started).  One
  // of the options below will change the value to TRUE when that happens
```


BEST COPY AVAILABLE

```
// the condition (!done) will become FALSE and execution will continue
// with the lines below the while loop block.

while (!done)
{
  customerNumber = 0;       // initialization for later use....
  // the following statement draws a menu on the screen
  cout << endl << endl << endl
       << SPACE << "*************************************************\n"
       << SPACE << "*           Mount Pleasant City Services        *\n"
       << SPACE << "*         *********************************     *\n"
       << SPACE << "*         Enter the Residential lot number      *\n"
       << SPACE << "*         for services to be billed             *\n"
       << SPACE << "*         or 9 to Quit                          *\n"
       << SPACE << "*                                               *\n"
       << SPACE << "*************************************************\n"
       << endl << endl;
  cout << SPACE << "             Selection? ->  ";
                          //  This is a user prompt

  cin.get (selection);   // we will read the number one character at a time
  while (selection == '\n')
     cin.get (selection);
  while (isdigit(selection))
       // as long as we are reading a valid digit
  {
    if (isdigit(selection))
    {
      digit = selection;       // automatically converts char to ascii value
        // the ascii value for character  '0' (zero) is 48 so we use
        // that value to convert the current value of 'digit' to the
        // correct number. the next line 'builds'
        // a customer number starting with the leftmost
        // digit.  For customer 127 for example we start with '1'
        //  customerNumber = 0 * 10 + 49 - 48    ->  1   then
        //   customerNumber = 1 * 10 + 50 - 48    ->  12  and finally
        //   customerNumber = 12 * 10 + 55 - 48   ->  127
      customerNumber = customerNumber*10 + (digit - 48);
    }
     cin.get (selection);
  }
  if (customerNumber == 9)
      selection = '9';
  if (selection == '\n') // we have a valid customer number
       selection = '1';

     // the switch statement is a  multi-way decision structure.  Execution
     // will continue with one of the statements below based on what value
     // is stored to the memory location called selection.
  switch (selection)
  {
   case '1': GetResidentData(customerNumber);
             break;
   case '9': done = TRUE;
             break;
   default : Error (1);
  } // end switch statement
    // if the user chooses 9 we exit the loop
    //- otherwise we run the requested function
    // and then return to the top of the while loop
} // end while loop statement
// For execution to continue here, the user has selected 9
cout << SPACE << SPACE << "Have a nice day!!";
return (0);
```


BEST COPY AVAILABLE

```
} // end function main
//*****************************************************************************
//  This function accesses the residential database to print the mailing
//  information for the invoice.  If the customerNumber is not valid, the
//  function calls the function Error and it reports the error condition
//  to the user.
//
int GetResidentData(int customerNumber)
{ ifstream addressBook;
  char customerName[25], address[25], city[15],zip[12];
  int  lotNumber = 0;
  addressBook.open("address.dat");  // put read marker at beginning of file
  if (!addressBook)
  {
    Error(3);
    return 1;
  }
  addressBook >> lotNumber;              // priming read
  // find this customer record  if it is in the file
  while ((lotNumber != customerNumber) && (lotNumber != 999))
  {
    addressBook.ignore(256,'\n');       // skip the rest of this line
    addressBook >> lotNumber;           // get another lot number
  }
  if (lotNumber == customerNumber)
  {
    addressBook.get (customerName,26);
    addressBook.get (address,26);
    addressBook.get (city,16);
    addressBook.get (zip,11);
    addressBook.ignore(20,'\n');
    // print the invoice heading
    invoiceFile << "Mount Pleasant City Services \n"
                << "235 Main Street \n"
                << "Mount Pleasant, North Carolina  12345-1000 \n"
                << "\n\n\n\n\n";
    invoiceFile << SPACE << SPACE << customerName << endl
                << SPACE << SPACE << address      << endl
                << SPACE << SPACE << city   << ", NC  " << zip
                << endl << endl << endl << endl << endl;
    invoiceFile << SPACE
                << "Please remit charges by the 10th of the month.\n\n\n";

    PrintCharges (customerNumber);
    invoiceFile << '\f' << '\n';
                    //outputs a form feed character so that the next
                    // invoice will print on a new page
  }
  else
       Error(2);
  return 0;
}  // end of function GetResidentData
//*****************************************************************************
//  a function designed specifically to report errors is a nice touch in an
//  application.  It allows reuse of some messages since the incomming
//  parameter is "hardcoded" by the programmer.  The default value message
//  might be encountered during program development but should never be
//  encountered by the end user - hence the message....

void Error (int condition)
{
  char anyKey;

  cout << endl << endl << endl
```

BEST COPY AVAILABLE

```
        << SPACE << "*****************************************************\n"
        << SPACE << "*           Mount Pleasant City Services            *\n"
        << SPACE << "*              Error Condition Alert                *\n"
        << SPACE << "*****************************************************\n";
  switch (condition)
  {
    case 1 : cout << SPACE
                  << " You have entered invalid data.  Please try again \n";
             break;
    case 2 : cout << SPACE
                  << "The customer number you entered does not exist \n"
                  << SPACE << "in the database. \n" ;
             break;
    case 3 : cout << SPACE
                  << " ***** Can't access input file! ***** ";
             break;
    default: cout << SPACE
                  << "The programmer is an idiot, an undefined error \n"
                  << SPACE << "has been reported!!\n";
             break;
  }  // end switch statement
  cout << endl << SPACE << "    - Press a key to continue";
  cin.ignore(1,'\n');
  cin.get(anyKey);
}  // end function Error
//*****************************************************************************
// You write this one...
void PrintCharges(int customerNumber)
{

} // end of function PrintCharges
```

BEST COPY AVAILABLE

REFERENCES

Almstrum, V. L. (1991). The relationship between pre-college mathematics and the undergraduate computer science curricula. Proceedings of the twenty-second SIGCSE Technical Symposium on Computer Science Education , USA, 23 (1), 124-129.

Ausubel, D. P. & Fitzgerald, D. (1961) The role of discriminability in meaningful verbal learning and retention. Journal of educational psychology 52 (5), 266-274.

Ausubel, D. P. (1963) Psychology of meaningful verbal learning. New York: Grune & Stratton.

Ausubel, D. P. (1968) Educational psychology a cognitive view. New York: Holt, Rinehart and Winston.

Bailie, F. K. (1991). Improving the modularization ability of novice programmers. Proceedings of the twenty-second SIGCSE Technical Symposium on Computer Science Education , USA, 23 (1), 277-282.

Bigge, M. L., & Shermis, S. S. (1992) Learning theories for teachers, (5th ed.) NewYork: Harper Collins.

Brilliant, S. S & Wiseman, T. R. (1996). The first programming paradigm and language dilemma. Proceedings of

the twenty-seventh SIGCSE Technical Symposium on Computer Science Education, USA, 28 (1), 338-342.

Campbell, D. T. & Stanley, J. C. (1963) Experimental and quasi-experimental designs for research. Boston: Houghton Mifflin.

Campbell, P. F. & McCabe G. P.(1984) Predicting the success of freshmen in a computer science major. Communications of the ACM , 27 (11), p.1108-1113.

Chinn, C. C. & Brewer, W. F. (1998) Theories of knowledge acquisition. In B. J. Fraser & K. G. Tobin (Eds.) International handbook of science education (pp 97-113) Great Britain: Kluwer.

Dale, N., Weems, C., & Headington, M.(1998) Programming in C++. Sudburry, Massachusettes: Jones and Bartlett.

Denman, R., Naumann, D. A., Potter, W. & Richter, G. (1994). Derivation of programs for freshmen. Proceedings of the twenty-fifth SIGCSE Technical Symposium on Computer Science Education, USA, 26 (1), 116-120.

Dyck, V. A. (1993). Emphasizing the process in delivering CS-1. Proceedings of the twenty-fourth SIGCSE Technical Symposium on Computer Science Education, USA, 25 (1), 127-130.

Ellis, R. (1985) Understanding Second Language Acquisition. Oxford: Oxford University.

Frazer, K. (1998) Comparing the impact of two

assignment-based teaching methodologies on student programming. Journal of computer science education 12 (1&2), p.21-26.

Gay, L. R. (1996) Educational Research: competencies for analysis and application. (5th ed.), Upper Saddle River: NJ: Prentice-Hall.

Genishi, C. (1988). Young children's oral language development. ERIC Digest PS-4-1988. Available : http://ericps.crc.uiuc.edu/npin/respar/texts/learning/lang.html.

Holt, R. W., Boehm-Davis, D. A., Schultz, A. C. (1987) Mental representation of programs for student and professional programmers. In Olson, G. M., Sheppard, S., Soloway, E. (Eds.) Empirical Studies of programmers: second workshop (pp 33-46) Norwood, NJ : Ablex.

Howatt, J. W. (1994) On criteria for grading student programs. SIGCSE Bulletin, 26 (3), p. 3-7.

Howe, A. C. & Jones, L (1998). Engaging children in science. (2nd ed.) Upper Saddle River, N.J.: Prentice Hall.

Jones, H. L. & Russell, J. M. (1979). Hierarchical learning paradigm. Journal of research in science teaching. 16 (6), p 489-499.

Kearsley, G. (July 3, 1998) Theory into Practice Database. Information Processing Theory (G. Miller) [On-line]. Available: http://www.mirrors.org.sg/tip/miller.html

Kerlinger, F. N. (1986) Foundations of behavioral research, 3rd ed. Chicago: Holt, Rinehart & Winston.

Konvalina, J, Wileman, S. A. & Stephens, L. J. (1983) Math proficiency: a key to success for computer science students. Communications of the ACM, 26 (5), pp 377-382.

Krashen, S. D. (1982) Principles and practice in second language acquisition. Oxford: Pergamon Press.

Lightbown, P. M.,Spada, N. (1995) How languages are learned, 5th ed. Oxford: Oxford University.

Long, T. J., Welde, B. W., Hollingsworth, J., Sitaraman, M. & Edwards, S. (1998) Providing intellectual focus to CS1/CS2. Proceedings of the twenty-ninth SIGCSE Technical Symposium on Computer Science Education 30 (1), pp 252-256.

Mayer, R. E.(1979) Can advance organizers influence meaningful learning? Review of Educational Research 49 (2), pp 371-383.

Mayer, R. E. (1981) The psychology of how novices learn computer programming. Computing Surveys 13 (1), pp 121-141.

Miller, G. A., (1956) The magical number seven, plus or minus two: some limits on our capacity for processing information. The Psychological Review 63 (2), pp 81-97.

Miller, L. A. (1981) Natural language programming: styles, strategies, and contrasts. IBM System Journal 20 (2), pp 184-215.

Perfetti, C. A. (1979) Levels of language and levels of process. In L. S. Cermak & F. I Craik (Eds.), Levels of processing in human memory (pp 159 - 181). Hillsdale, NJ: Erlbaum.

Price, G. E. (1996). Productivity environmental preference survey. Lawrence, KS : Price.

Sevier, R. C. (1995). Whole-sentence versus isolated-words: the efficacy of two reading intervention for "at-risk" prereading 6-year-olds. (Doctoral dissertation, University of Southern Mississippi, 1995) Dissertation Abstracts International.

Shavelson, R. J. (1996). Statistical reasoning for the behavioral sciences. (3rd ed.) Needham Heights, Mass: Allyn & Bacon.

Watson, G. B. & Glaser, E. M.(1994) Watson-Glaser Critical Thinking Appraisal Form S Manual. San Antonio, TX: The Psychological Corporation.

Wiedenbeck, S. (1986). Processes in computer program comprehension. In Soloway, E., Iyengar, S. (Eds.) Empirical studies of programmers (pp 48-57) Norwood, NJ : Ablex.